technische universität
dortmund

Master's Thesis

Aggregation on Computational Memory

Dennis Martin Bednarek

April 2024

Supervisors:
Prof. Dr. Jens Teubner
Roland Kühn, M.Sc.

TU Dortmund University
Department of Computer Science
Databases and Information Systems Group
http://dbis.cs.tu-dortmund.de/

Contents

Chapter 1

Introduction

1.1 Motivation

The advancements in memory technology in the past decades have allowed for the widespread usage of large and cheap memory units in personal, commercial and scientific computing. However, even though memory capacities and throughput have increased significantly over the past decades memory latencies have remained almost stagnant in comparison[5]. As data intensive applications now seek to process Terrabytes of data, time is therefore increasingly spent on the movement of data[1].
One alternative paradigm, that has recently gained traction is that of processing in memory(PIM)/computational memory, which promises to reduce times spent for waiting on memory cells, by placing some processing capabilities close to the memory[13].

While the most accessible way of using the parallelism provided by modern multi-core CPUs is the use of threads, which are implemented for almost all hardware platforms, threads obfuscate the underlying architecture, in the worst case incurring additional overhead through expensive context switches [17]. Modern task based frameworks such as MxTasking [11] aim to avoid these pitfalls by enabling the programmer to develop their program in a manner, that is considerate of architectural characteristics.

1.2 Goals of this thesis

The main objective of this thesis is to reevaluate the previous scientific observations regarding the database operation of aggregation in a current technological context. For this purpose, aggregation with regard to various types of workloads will be examined utilizing modern hardware, with both a traditional architecture i.e. a multicore CPU, as well as a specialized architecture i.e. a hybrid system using an accelerator optimized for database-workloads. The accelerator chosen for this thesis is the UPMEM PIM system.

Both architectures will be utilized using a task-based framework, specifically MxTasking, in order to evaluate the effectiveness of a task based framework on this computational memory platform.

Additionally, given the novel architectural environment given by the UPMEM PIM system this thesis will focus on exploring various approaches to implementing aggregation on this system and discuss benefits and shortcomings thereof for different types of workloads.

1.3 Structure of this thesis

Chapter 2 will introduce the necessary concepts for this thesis starting with the algorithmic problem of aggregation and its previous solutions in the literature, followed by an introduction to the concept of task-based frameworks leading into the MxTasking framework. Finally, the technology of Computational Memory will be introduced. Chapter 3 will discuss the extension of the MxTasking framework, onto the UPMEM PIM system. The implementation of a set of aggregation algorithms utilizing the UPMEM PIM system is discussed in chapter 4. Chapter 5 is dedicated to the discussion of the experimental setup and the retrieved results from both the UPMEM-based implementation as well as the purely CPU-based implementation which has been created for comparison. Finally, chapter 6 will summarize the results of this thesis and give a brief overview of possible future work on the subject matter.

Chapter 2

Fundamentals

2.1 Aggregation

The emergence of the first multi-core processors brought the potential for an increase in performance for the algorithmic problem of aggregation, while also requiring more sophisticated and more specialized algorithms to fully utilize the available processor cores[6]. Previously the existence of the cache-hierarchy had strictly been beneficial for workloads with smaller group-by-cardinalities or workloads with keys that are clustered in one way or another, because the small hash table size required to store the aggregation results could fit into the smaller caches resulting in a reduced memory access time when aggregating. The emergence of multi-core processors introduced the new factor of contention into this algorithmic problem[6]. This led to the invention of a variety of different algorithms meant to alleviate the performance decreases caused by contention, of which four key approaches will be discussed in the following.

2.1.1 Sorting based and hash based aggregation

While the majority of aggregation algorithms that are going to be discussed in this thesis utilize the concept of hashing, sorting also is an option for implementing an aggregation algorithm. In spite of the inherent cache friendly behavior of some sorting algorithms, the fact that even the fastest sorting algorithms have a worst case runtime complexity of $O(n \cdot logn)$ whereas hash based algorithms can achieve runtimes in $O(n)$ as long as the used hash table is sufficiently large, and the hash function has been chosen adequately, has generally led to the dismissal of these algorithms [6].

2.1.2 Independent table aggregation

As implied by its name, this algorithm utilizes one independent table each for each processor-core in the system. Only when a core is done with its given workload is it necessary to merge the resulting hash table into a centralized hash table.

The big advantage of this approach is that it entirely avoids contention, excluding the merge-phase, which depending on the group-by-cardinality may be very short.
While this approach does avoid the overhead caused by contention and the necessary synchronization coming along with it, it simultaneously runs into the problem of requiring significantly more memory to store its hash tables. While this memory requirement typically is not a big problem for today's in-memory databases as RAM-capacities can reach hundreds of GB this attribute of this approach can quickly lead to the hash tables no longer fitting into the Last Level Caches (LLC) of the given system [6]. This typically means that other approaches outperform the independent hash tables when the group-by-cardinality induced hash table size reaches a fraction of the size of the LLC because of the comparatively slow RAM-access times[6].
Conversely, the nonexistent overhead for small group-by-cardinalities causes this approach to be by far the most performant for small group-by-cardinalities[6].

2.1.3 Shared table aggregation

This approach is effectively the opposite of the independent table algorithm, as it simply uses one hash table which all processor cores work on at the same time. This means however that in order to avoid race conditions, some kind of synchronization method is necessary.
The two main techniques for synchronizing the access to a shared variable are locking and atomic-instructions. Due to the independent nature of the individual entries in the hash table, atomic-instructions can be easily utilized for this purpose, while outperforming locking due to the lower overhead associated with its instructions[6].
Alternative approaches to the utilization of shared hash tables for aggregation have been inquired in the literature such as the idea of contention detection, where aggregation would primarily occur in a shared hash table until contention was detected by the inability to access a key in the table for one thread[18]. The key for which the contention was detected would then be evicted for every table into a private hash table, thereby avoiding further slowdown on this potentially contentious key.

2.1.4 Hybrid aggregation

While the shared table algorithms are typically able to outperform the independent table algorithm for large group-by-cardinalities, this may not be the case if the distribution of keys is so severely imbalanced that it induces a large enough amount of contention on some key or group of keys, such as is the case for heavy-hitter-distributions.
Given such a workload it would be beneficial to separate common keys and to store them in an independent hash table for every thread to avoid the contention on them, while still using a shared hash table for the remaining keys in order to limit the total size of all hash

tables such that the cache doesn't get trashed for medium group-by-cardinalities as would be the case for the independent hash table algorithm [6].

The hybrid aggregation algorithm implements this idea by utilizing small independent hash tables for every thread which are usually sized to fit into one of the cache levels, while keeping track of some metric for determining which keys to keep in this private table, usually this is the access frequency or last access time [6]. When the private hash table is full, the key with the lowest score with regard to the metric will be evicted into the shared table and possibly merged with previous values for that key [6].

While this algorithm is particularly well suited for the aforementioned imbalanced key distributions, its performance for more balanced distributions is slightly worse than that of the two other algorithms [6]. This is because using the private hash tables causes an additional overhead, in particular for large group by cardinalities, where the majority of keys will have to reside in the large shared hash table, but still need to be checked in the private hash table [6].

2.1.5 Partitioning based aggregation

As described above, the performance of aggregation algorithms is heavily dependent on the distribution of the keys in the given workload, because of its influence on both the contention and the potential cache misses caused by it. By performing a partitioning step before executing the aggregation it is however possible, to both avoid the contention that would be caused by the access to common keys as well as avoiding a substantial amount of cache misses caused by other algorithms[18]. This however comes at the substantial cost of having to perform the partitioning step which, is the main performance bottleneck of this algorithm and has therefore seen a large amount of scientific investigation into optimizing it for varying workloads[18][7][12].

Partitioning is performed by computing a hash function on the key of every tuple, and using it to segregate the data into partitions which only contain a limited but exclusive range of the possible keys[18]. The size of each range is chosen such, that the hash table or each partition can fit into some cache-level, thereby ensuring no cache-misses during the aggregation step[18]. Due to the large overhead which is caused by the partitioning step this algorithm is outperformed for small group-by-cardinalities by the independent algorithm, though there are hybrid approaches combining both. For very large group-by-cardinalities however, this approach outperforms all other approaches, as the partitioning overhead is smaller than the RAM access overheads incurred by other algorithms for the majority of all tuples[18].

2.2 MxTasking

MxTasking[11] is a recent example of a task based framework for application development. It realizes a system for managing and scheduling tasks, as small batches of worker, which can be scheduled onto a specific core while being executed without interruption. These tasks can also store additional information to enable the efficient scheduling of the tasks[11]. This information includes the priority of a task and its target.
While options exist to use the MxTasking framework by directly annotating a specific processor core to a task as its target, an additional option is provided to specify a data object as the tasks target instead. In doing so the MxTasking framework provides the programmer with an automated way to structure the scheduling of a programs tasks around the data of the program on the granularity of individual tasks[11].
In order to allow for this automated data oriented scheduling to take place, MxTasking allows the programmer to annotate such data objects. The options for these annotations include the following:

1. The target, i.e. a processor core or a NUMA node
2. The expected access frequency
3. The read write ratio
4. The preferred synchronization protocol

These annotations are mainly used by the framework to provide a way to automate the synchronization of each task. As resource objects may be of any size and as the synchronization method is applied for the duration of a task's execution, the available synchronization protocols do not include atomic operations and instead support approaches linked with locks, such as spin locks, mutexes and write-specific locks, though it is also possible to not use the runtime's synchronization[11].
Additionally the information available to the framework about what data a task intends to use during its execution, enables it to issue prefetch instructions to the responsible processor core, as the framework controls the order of the execution of tasks for each core. In doing so some memory latencies may be hidden[11].

2.3 Computational Memory

Computational memory as a term can refer to two different ideas. One idea expressed by the term, is the implementation of computations directly in memory, i.e. having the necessary circuitry for some computations not only reserved for the registers inside of a processing core, as it is historically common for computers, but rather to provide this infrastructure for what is considered the memory of the given computer, which is referred to as processing using memory(PUM)[13]. The second idea expressed by the term, is moving the infrastructure needed for computations closer to the memory, which is referred to as processing near memory(PNM)[13]. One possible approach to achieving that would be to associate every chunk of memory of some given size with some kind of processor-core which is located in proximity of its chunk of memory, which is the approach taken by the UPMEM PIM system[16].

2.3.1 Architecture

The UPMEM PIM-system takes the shape of commonly available RAM-modules called DIMMs of which depending on the mainboard of the used computer up to 20 may be used at a time[16]. Each of these DIMMs consists, depending on the version, of either one or two ranks, which are the smallest units which can be invoked independently[16]. Each rank is made up of 8 chips, which each contain each 8 memory banks with a capacity of 64 MB and every memory bank has a dedicated DRAM processing unit (DPU) associated with itself, which enables it to independently perform computations[16].

Each DPU consists of four components, which are the processing core itself, the 64 MB memory bank which is called main RAM (MRAM), a 64 KB scratchpad memory which is called working RAM (WRAM) and a 24 KB instruction memory called instruction RAM (IRAM)[16].

Each DPU is a 350 MHz in-order single-core processor and is equipped with 24 register sets enabling the use of up to 24 hardware-threads, which are called tasklets, at a time. Notably because a DPU is a single-core processor every tasklet on a DPU shares the same pipeline[16].

The DPUs utilize a 16-step pipeline, however only the last 5 steps of an instruction may be executed at the same time as the first 5 steps of a subsequent instruction from the same tasklet[8]. Therefore, the maximum performance of an UPMEM DPU can only be achieved when at least 11 tasklets are executed at a time[8].

With the exception of MRAM accesses every assembler-instruction that can be executed on a DPU needs exactly 16 cycles, which means that for a full pipeline one instruction can be executed in one cycle on average[8]. However, while the supported instruction set does contain most simple operations, such as additions, subtractions, logical operations and bit operations both for signed and unsigned 32-bit integers, there is no hardware-support for multiplications, divisions or any kind of floating point operations[16]. These operations are instead implemented as a sequence of the available assembler-instructions which makes their execution significantly slower when compared to the relative speed on other hardware. [8] therefore recommends to avoid using these instructions when possible.

While each DPU has the three memory segments IRAM, WRAM and MRAM at its disposal, only the WRAM is directly accessible from the given DPU[16]. As with every

DPU-assembler-instruction a WRAM-access takes 16 cycles[16]. However, as programs on the DPUs can only utilize the full available performance of the hardware if at least 11 tasklets are running at the same time and as all three memory segments are shared among the tasklets of a DPU some kind of synchronization is needed to avoid race-conditions when accessing the same memory with multiple tasklets. For this purpose there are 56 hardware-mutexes available which may be used directly or indirectly through various libraries to implement more advanced synchronization patterns[16]. While a lock/unlock-operation is significantly faster in relative terms than on conventional hardware, as no actual communication with another processor is necessary, a singular lock-call is implemented in 1 assembler-instructions in addition to being implemented by spinning, i.e. taking up the pipeline when waiting on another thread[16].

As mentioned before access to the MRAM is treated in a special manner by the DPU as doing so removes the calling tasklet from the pipeline until the requested memory has been retrieved. Additionally, only one MRAM access may be going on at a time, meaning that other tasklets issuing MRAM-access instructions will also suspend their execution when waiting for the previous tasklet to finish its access, though this also means that access to the MRAM is functionally atomic[16]. Furthermore, access to the MRAM must happen both aligned on 8 Bytes and in multiples of 8 Bytes reaching its maximum at 2048 Bytes per transfer[16]. Because every access to the MRAM incurs an access-latency in addition to the per-byte-constant transfer time, throughput increases for larger transfer-sizes, as the access-latency becomes a smaller component of the time [8]. This means that random access on small byte amounts incurs a significant penalty compared to large sequential reads or writes from/to MRAM, as a single read takes 77 cycles whereas a single write takes 61 [8].

Finally the IRAM of a DPU is not accessible for programs to work in, but instead stores the entirety of the code that can be executed on a DPU during a given invocation. While it is integrated in such a manner, that instructions can always be loaded into the appropriate register on time [16] its organization also imposes a hard limit on the complexity of the code that can be executed during one given invocation of a DPU. With one instruction taking up 6 Bytes this limit is reached at 4096 assembler-instructions [8].

One notable characteristic of the UPMEM PIM system is that every DPU has only access to its own memory, i.e. its IRAM, WRAM and MRAM and that it by itself can neither communicate with other DPUs directly nor do so with the CPU[16]. As the DPUs are treated as an accelerator system for the CPU to invoke, everything the DPUs do must be called on by the CPU through the provided library.

2.3.2 Supported Software

The SDK provided by the UPMEM company includes a compiler for the DPU programs, the required libraries for implementing a host application, as well as debuggin tools for both[16]. The host application can be written in C, C++, Java or Python, however, the low level API is only implemented in C code. The API provides access to all operations necessary for sending and receiving data from the DPUs as well as for their invocation[16]. DPU-programs can be written in C, the specialized assembler-language or a combination of both. The libraries implemented for the DPU-programs include most of the standard C libraries such as the stdio.h and the stdlib.h functionality for transferring data between the WRAM and MRAM is provided in the mram.h library[16].

Due to the performance criticality of implementing multi tasklet programs, additional libraries are provided by the UPMEM SDK, for synchronizing the access to the memory. These include a mutex, barrier, semaphore and handshake implementation, which internally use the provided 56 hardware mutexes[16].

2.4 Related work

[6] , [18] and [14] have discussed aggregation on multi-core processors, for various input distributions.
[8] discussed the performance of the UPMEM PIM system for a variety of general workloads, while [3], [4] and [2] discussed and evaluated various use cases of UPMEM with regard to database workloads. [9] has evaluated the UPMEM PIM system for a selection of database operator including aggregation, while considering one particular approach for distributing the hash tables across the available DPU memory, that is similar, to the hybrid algorithm.
Finally, [15] has already implemented support for the use of remote accelerators for the MxTasking framework, while however staying limited to the use of conventional CPU based architectures. For that purpose they have particularly focussed on the granularity of the transferred tasks.

Chapter 3

A task-based approach to utilizing an UPMEM PIM system

As alluded to in section 1.2 the implementation of the aggregation experiments which this thesis focuses on, is based on the MxTasking framework introduced in subsection 2.2.
As discussed in section 2.3.2 the UPMEM PIM system cannot operate independently of a host program as the individual DPUs only have access to their own private memory segments and cannot communicate among each other. Additionally, the programs that run on the UPMEM PIM system must be compiled with a specialized compiler provided in the UPMEM SDK and then transferred into the IRAM of the targeted DPU set. As the MxTasking framework is mainly written in C++ it is compatible with the host libraries which are available in both C and C++[16]. Therefore, the following sections are going to discuss the implementation of the extension of the MxTasking framework for the UPMEM PIM system.

3.1 Software architecture of the host side

As the code of the host side is an extension of the original MxTasking framework, a brief rundown of it shall be given in the following subsection.

3.1.1 Original MxTasking software architecture

As can be seen in figure 3.1 the MxTasking framework is centered around the runtime class, which is a utility class that provides access to everything necessary for using the framework. As discussed in subsection 2.2 the MxTasking framework does not only automate the scheduling of tasks, but also provides options for doing so while basing the scheduling on data used by each task.
Therefore the two main objects of the MxTasking framework are the task and the resource, which are both abstract classes that are managed by the subsystems of the runtime and

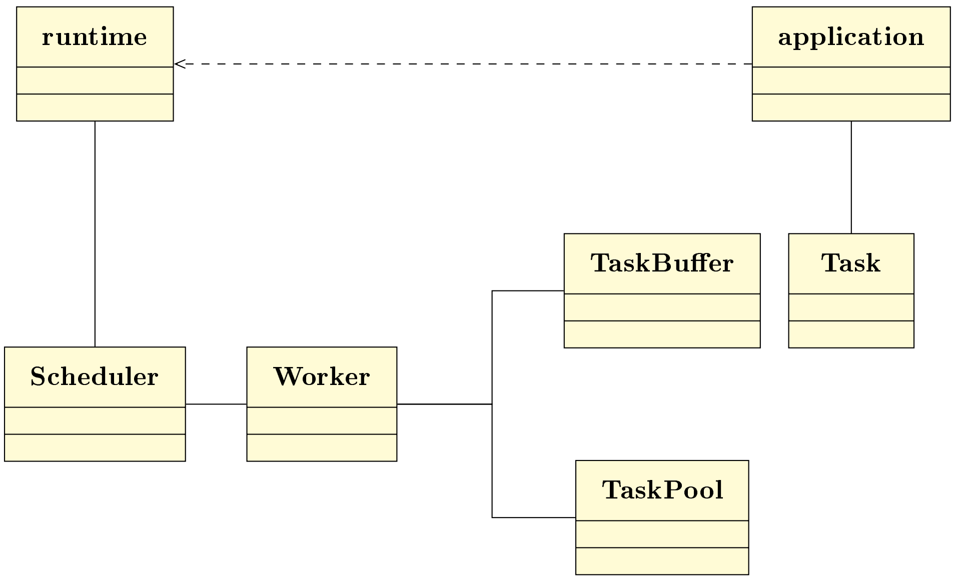


Figure 3.1: The central components of the original MxTasking framework [11].

need to be overwritten by the programmer to use. The main operations that the runtime supports with regard to these two primitives is their creation, annotation and deletion, which is managed by one specialized allocator each and the dispatching of the tasks onto the cores of the computer using the Scheduler class [11].

The main purpose of the Scheduler class is the managing of the individual processor cores of the given system, which are abstracted by the Worker class. The Scheduler therefore stores one Worker object for each processor core used by the runtime and each Worker executes one thread which is permanently pinned to a processor core. Each Worker owns a TaskPool (made up of different underlying queues for synchronized and unsynchronized access) and a TaskBuffer. Tasks that get scheduled to a Worker by the Scheduler are first added to its TaskPool. When the setup of the program is finished and the Workers are launched, the Worker transfers the oldest tasks with the highest priority into the buffer which has a static size, where they are prepared for prefetching and executed in a strict order. Tasks must return a TaskResult upon completion, which may either spawn followup tasks, contain information about how to handle the tasks memory or instruct the runtime to shut itself down. So the process of Workers executing tasks from the TaskBuffer and fetching new tasks from the TaskPool which new tasks can be scheduled to is repeated until a task returns the shutdown signal[11].

The lifecycle of a task therefore consist of the following steps:

1. The creation of the task

2. The annotation of the task e.g. with a core or a resource which has been created beforehand

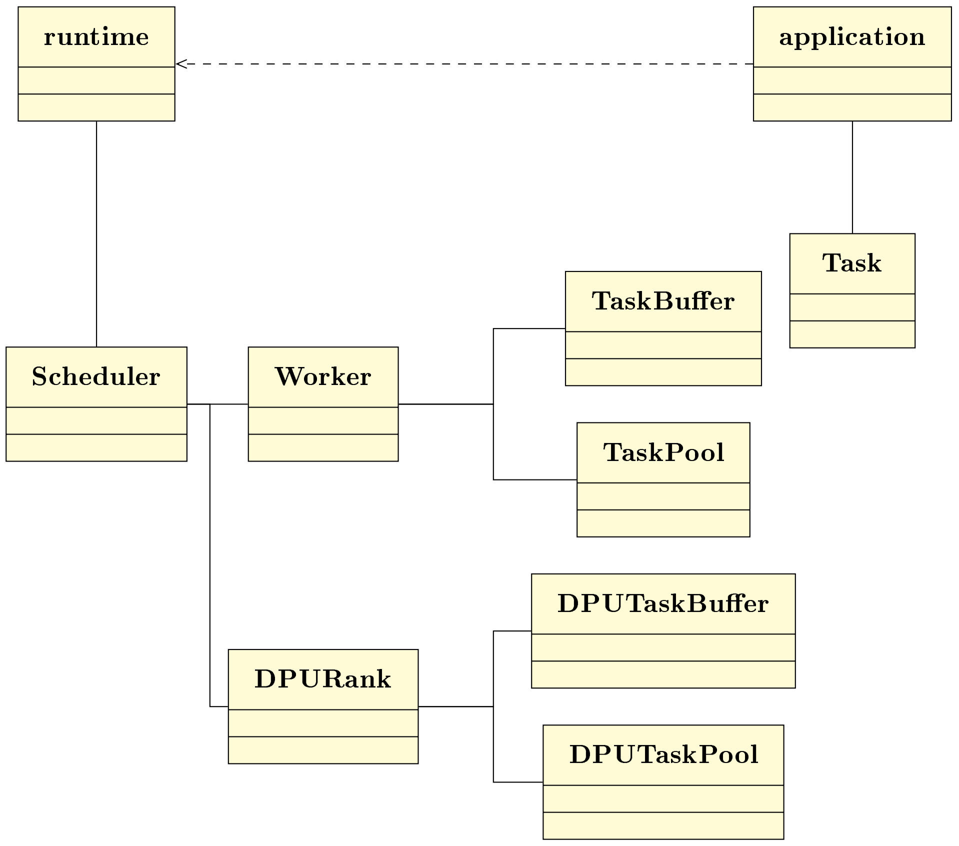


Figure 3.2: The central components of the extended MxTasking framework.

3. The dispatching of the task, which leads to its scheduling and execution

4. The deletion of the task

3.1.2 Extension of the MxTasking software architecture onto the UPMEM PIM system

In order to extend the MxTasking software such that it supports scheduling tasks onto the UPMEM DIMMs it is crucial to consider how the individual components of UPMEM may be addressed by the programmer.
While each DPU can run up to 24 so called tasklets, which aside from sharing the same WRAM and MRAM can execute code independently, these tasklets cannot be invoked independently. Furthermore, not even the 2560 DPUs available in a full UPMEM installation can be invoked independently. Instead, DPUs are grouped together in groups of 64 which are called ranks and which are the smallest unit that can be invoked independently and with different code[16].
In order to incorporate these hardware limitations the following decision with regard to the framework's software architecture have been made. In order to provide a maximum

amount of precision to the programmer with regard to the location of the execution of a task on the UPMEM system, each tasklet is functionally treated as the equivalent of a worker on the CPU. Therefore, tasks may be scheduled to specific tasklets.
However because only one DPU set may be allocated and invoked per rank the scheduler uses the DPURank class as the chosen abstraction. By doing so a similar approach as the one taken for CPU tasks may be used for scheduling to the UPMEM DIMMs. While tasks may be scheduled to the DPUs on a granularity of their tasklets the Scheduler merely computes the rank that tasklet belongs to and then passes it on to the appropriate DPURank object.
The DPURank object contains an array of DPUTaskPools where each represents one of the DPU workers i.e. tasklets. When the scheduler is launched the DPURank object will retrieve as many tasks from the pools as possible and store them in the DPUTaskBuffer, which is only present once per DPURank object. In the DPUTaskBuffer object the tasks are converted to a format which is appropriate for the execution on the DPUs, that is, an 8 Byte object, which only contains an 1 Byte ID that identifies the type of the task and a 4 Byte address used to locate the address of an optional associated resource on the local DPU. After having been converted to this simplified format the tasks are stored into the buffer that is going to be used to transfer them to the DPUs of the rank. The structure of this buffer depends on the chosen DPU runtime implementation, and is going to be discussed in depth in section 3.2.
In order to accelerate the address computations for the task batches both on the CPU and DPU, the number of concurrently working DPU workers, was set to 16, as it has the advantageous characteristic of both being a power of two between the minimum required 11 tasklets and the maximum 24.
During execution each rank is managed by one thread, however as the actual actions of these threads are mostly limited to waiting in a loop for the finishing of the ranks execution and preparing the tasks for transfer, these threads are not pinned to a specific core. The steps taken in the loop are as follows:

1. The rank's thread checks whether a signal has been sent to invoke the DPU rank. If not the loop repeats.

2. The thread fetches as many tasks as possible from its DPUTaskPools and converts them to the appropriate format for the DPUTaskBuffer. Then it transfers them to the MRAM of the ranks DPUs.

3. The thread invokes the DPU rank using the synchronized method, i.e. it waits for its completion.

4. Once the thread determines, that the rank is done, it transfers the followup task buffer from the MRAM and spawns the appropriate CPU tasks.

5. The thread returns to step 1.

To enable a more energy efficient approach, the active waiting of the ranks could be replaced with calls that enable the operating system to use some set amount of processing time somewhere else, however as this thesis is focused on the achievable performance such considerations have not been made.

3.1.3 DPU task definition and compilation

So far this thesis has discussed the procedure for distributing and transferring tasks to their designated workers on the DPUs. This section is going to introduce the procedure for defining and compiling the code of these tasks to prepare them for their execution on the DPUs.

```
bool task1(uint32_t resource_pointer)
{
        do_something();

        return true;
}
```

Listing 3.1: Example of an empty DPU task. As the address space of a DPU is less than 4 GB the 32 bit integer may be cast to a pointer if needed.

As the code that runs on the DPUs cannot be simply compiled as part of the host application, but instead needs to be separately compiled and transferred to the targeted DPU set's IRAM, additional steps need to be taken so that tasks may be spawned on the CPU at runtime and executed on the DPU. The runtime was therefore extended to support the definition of tasks and resources on the DPU as well as the compilation of those into a functioning DPU program. However, as the DPUs do not support the object-oriented features of C++, tasks are defined as functions of the shape seen in listing 3.1, whereas resources are defined as simple structs.

In order to define a DPU task, one must begin with the same procedure as when defining a task meant for the execution on the CPU, i.e. overwrite the abstract task class provided by the MxTasking framework. However, unlike a CPU task, the actual execute method may be left empty as it will not be used. Instead, the main usage of the data object associated with the class is to be registered at the DPUCompilationUtility, which is responsible for generating and compiling the code that is going to be executed on the DPUs.
In order to do so, two methods have been added to the original task class definition, which are a setter and a getter for the *dpu_task_id*, which is the one byte ID meant to identify the task type on the DPUs. The overwriting task class must therefore use a class attribute, and make it accessible through its setter and getter, which it needs to overwrite

appropriately. By doing so it is possible to pass an instance of the overwriting Task to the DPUCompilationUtility which will then generate a new ID and set it for that task, as well as storing it internally for code compilation.
When an instance of that task is later created and spawned, the scheduler will first check the annotation, which distinguishes the execution location, i.e. CPU or DPU in order to either pass it on to a CPU worker, or to the annotated DPURank. The annotation used for denoting a DPU worker is the same 2 byte attribute as that used for annotating the CPU core. This is because the maximum number of workers in the currently available version of UPMEM is $24 \cdot 64 \cdot 40 = 61440 < 65536 = 2^{16}$.
In case of a CPU annotation the process described in section 3.1.1 is followed.
In case of a DPU annotation the DPURank will append the Task object to the appropriate DPUTaskPool. When the rank is ready to be invoked, the least recently spawned tasks in the pool will be transferred to the DPUTaskBuffer, which utilizes the getter for the *dpu_task_id* to determine which type of task should be represented, as the DPUTaskBuffer converts the previous relatively complex task object to an eight byte struct, containing the one byte task type as well as a four byte pointer to the DPU resource, which may be assigned via the resource annotation.
Notably however, it is possible to overwrite a task class and to define functionality for both the CPU side and the DPU side. The version which is going to be actually executed then depends on the annotations that were used for that class at runtime.
Creating code for the DPU that is compatible with the MxTasking framework functions in the following way. When registering a task it is also necessary to provide the name and code of a C function as a string object. Both will then be used in combination with the registered ID to automatically be written into a template for the DPU runtime code, by adding them to the switch case in the main runtime loop on the DPU. In addition to defining tasks this way it is possible to register definitions of any kind in the appropriate DPU compatible C code by just creating a string with that code and using the register_declaration method provided by the runtime.

3.2 Software architecture of the DPU side

Unlike the host side of the application which will usually be run on a modern CPU with a frequency of a couple GHz, the DPUs used for the experiments of this thesis run at a frequency of 350 MHz. In addition to that it is necessary to execute multiple tasklets which share the same pipeline in order to fill the pipeline and take full advantage of the available 350 MHz, as discussed in section 2.3.1.
Both these characteristics make it so, that implementing a complex runtime system on the DPUs would lead to a comparatively significantly larger overhead than on a CPU. Therefore, in order to avoid burdening the system with this overhead the architecture

of the tasking runtime on the DPUs has been kept relatively simple as the majority of operations on the tasks such as the scheduling are managed on the host side as discussed in section 3.1.

A slightly simplified code segment of the runtime implemented on the DPUs can be seen in listing 3.2. As can be seen there, the DPU runtime is similar to the host runtime in that each worker iterates over a buffer of tasks which it has been assigned and executes them, where after it proceeds by retrieving new tasks. The differences are as follows:

```
main()
{
        if (worker.id == 0)
        {
                init_runtime();
        }
        do
        {
                load_next_task_batch_from_mram(worker.id);

                for (task in current_task_batch())
                {
                        switch (task.type)
                        {
                                case 0:
                                        shutdown_worker();
                                case 1:
                                        execute_task_type1(task.resource);
                                case 2:
                                        execute_task_type2(task.resource);
                        }
                }
        }
        while (next_task_batch_exists(worker.id))

}
```

Listing 3.2: Pseudocode for the central code executing on each worker of each DPU

- Tasks are not fetched from a queue but from an array stored in the MRAM
- As DPU code may only be written in C code, the different task types are not determined by polymorphism but instead by a switch case over the task type which has to be explicitly stored in the task struct
- Neither support for prefetching nor automated synchronization methods are implemented, to avoid the associated overhead

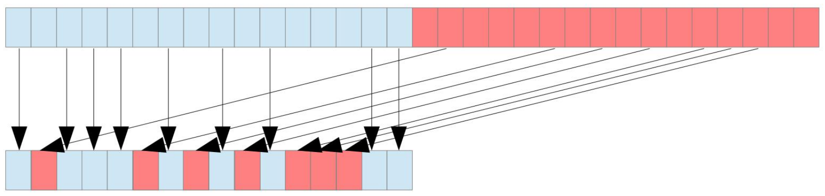


Figure 3.3: The structure of the task buffer used for the simple runtime implementation. The structure on top is the MRAM task buffer, from each worker fetches the next batch into the WRAM task buffer(bottom).

3.2.1 Alternative versions of the DPU runtime

As can be seen in the listing 3.2 the main function of the DPU runtime, is the retrieval and execution of the tasks for each DPU worker. As the tasks are represented by data in the shape of the task buffer, their movement competes with the movement of the actual application data, over the MRAM bandwidth available on each DPU. Additionally, the space required for storing the current batch of tasks competes over the limited WRAM space with the requirements of the application. Therefore, three approaches for implementing the runtime on the DPU have been implemented, which aim to minimize the performance overhead caused by the runtime.
In order to do so, these approaches attempt to use the maximum possible transfer size for the task-transfers, while keeping the size of the task buffer in the WRAM small, such that enough memory remains for the application data.

Simple runtime

The simple runtime represents a naive approach to implementing the runtime. It is equivalent to the runtime main loop represented in listing 3.2. This means, that every DPU worker simply fetches new task batches from the MRAM task buffer, until a task contains the 0 type or the end of the MRAM buffer is reached, at which point the worker aborts its execution. While this implementation requires no synchronization between the individual workers, it has the disadvantage, that every worker needs its own WRAM buffer. If this buffer's size is chosen to be the optimal transfer size of 2048 bytes all 16 workers will need to store one such buffer totaling in a memory requirement of 32 KB, thereby taking up half of the available 64 KB of the WRAM. As this would leave far too little space for the application data, as memory is still needed for the stacks, the decision has been made to instead aim for a transfer size of 1024 bytes and to fit all 16 workers in there (i.e. 64B per worker and transfer).

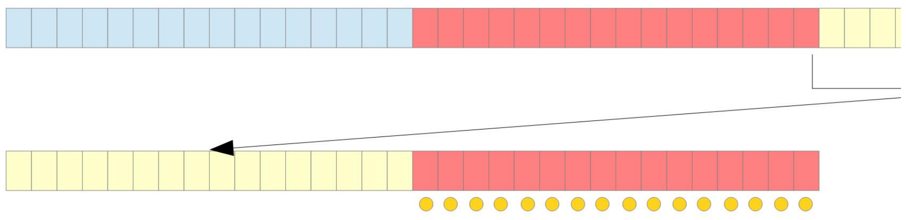


Figure 3.4: The structure of the task buffer used for the two buffer runtime implementation. The WRAM task buffer(bottom) has space for two task batches, which each have 16 task segments, where each segment contains the eight tasks belonging to one worker. New tasks from the MRAM(top) are only fetched if no more workers (yellow dots) are active in the given half of the task buffer.

In doing so a middle ground approach has been taken, which prioritizes leaving sufficient space for the application data, while allowing for a non synchronized task transfer, that cannot profit from the optimal transfer sizes thereby suffering from a performance decrease in that regard.

Two buffer runtime

An alternative approach to implementing the DPU runtime, is to discard the notion of not synchronizing the task transfers, and to thereby gain the ability to use shared, but larger transfer buffers(i.e 1024B), which do not take up as much total space, due to there being fewer of them. To do so this approach uses two buffers, which each interleave the tasks for the different workers, as can be seen in figure 3.4. Unlike the first implementation, this version, does not rely on the worker tasklets for fetching the task data. Instead, a separate $17th$ task loader tasklet is used which executes a different main loop, in which it waits on a semaphore and then loads in a new batch of tasks. This is because the semaphore is internally implemented by actually removing waiting tasklets from the pipeline until they are woken back up, thereby preventing this tasklet from taking up the pipeline unless it is executing its loading code[16].
When a DPU is launched, one worker will fetch the first two task batches, which contain tasks for all 16 workers, after which the workers will start their main loop, where they iterate over the tasks that belong to them in the buffer. However, whenever a worker finishes the last task of a batch it must check whether it was the last worker to leave the batch and if it was it is responsible for waking up the task loader tasklet which will then load in a new batch into the space of the older batch. When a worker is so fast, that it finishes all tasks from both current WRAM batches, before the older batch has been replaced it must remove itself from the pipeline, by waiting for a semaphore as well, which will later be woken up by the task loader tasklet.

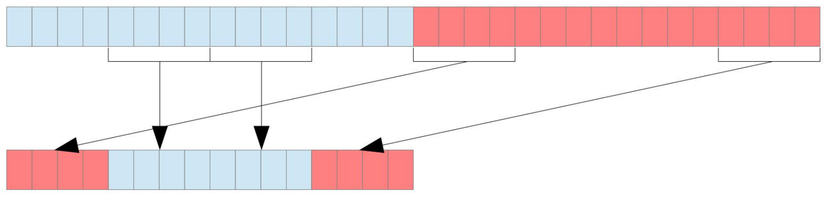


Figure 3.5: The structure of the task buffer used for the grouped worker runtime implementation. The buffer has space for four task batches, which each have four task segments, where each segment contains the 32 tasks belonging to one worker. The top shows the MRAM task buffer and the bottom the WRAM task buffer. Notably, each memory segment here is four times as large, as in the previous illustrations.

While this implementation allows for the use of the largest transfer sizes, while reducing the required memory down to $2 \cdot 2048B = 4KB$ from $16\dot{2}048B = 32KB$ (though in the experimental evaluations only 1024B are used per batch) for doing so, limiting the number of currently active batches down to two, means that very fast tasklets are forced to wait on the slowest tasklet. However, as 16 workers are used, this leads to an automatic balancing mechanism, because the remaining workers will take up a larger part of the pipeline during these intervals. While the synchronization overhead for each worker is limited to checking whether it was the last worker to finish working on a task batch and checking whether the task batch it worked on was the older one, this overhead is new if compared to the simple runtime implementation.

Grouped worker runtime

While the two buffer runtime implementation presents a relatively low overhead approach to implementing a synchronized task fetching system, not relying on synchronizing all workers for retrieval might be beneficial if the workload distribution is simultaneously hard to predict and unbalanced. The third approach to implementing the DPU runtime therefore organizes the 16 workers into groups of 4, which each use a separate $2048B$ WRAM buffer for their tasks. This however requires a different structure of the WRAM buffers, which can be seen in figure 3.5. The process of execution works as follows:

One worker for each group loads in the first task batch, where after they start working on the tasks. When a worker is done with its tasks in the batch, it checks whether it is the last one to finish.

If it is not, it removes itself from the pipeline by attempting to acquire a semaphore.

If it is, it loads in the next task batch for the group and wakes up the waiting members.

This implementation, has the benefit of having the groups operate independently, thereby requiring four very slow workers across the groups to slow the execution down, thereby decreasing the likelihood of that happening. Additionally, as each group has a larger buffer at its disposal, the instruction count for synchronization is reduced if compared to the two buffer implementation. It should be noted that this leads to a bigger memory requirement. The main idea of this implementation, is that, as four groups of four are present, the majority of the time 16 workers are going to be active at a time. When a group is stopped only four workers are missing from the pipeline for the duration of the task transfer, thereby still allowing for a full utilization of the pipeline with 12 workers. If however, more than one group is waiting for its task transfer at the same time, there is a risk of the pipeline not being full.

The experimental evaluation of these runtime implementations is going to take place in chapter 5, as well as an experiment regarding the overall overhead of the tasking system in general.

Follow-up tasks

In order to provide an easily accessible structure for the communication from the DPUs to the CPU a follow-up task mechanism was implemented on the DPUs tasking runtime. This simply means, that an additional task buffer with 8 task slots per worker is present in the WRAM of each DPU, and to which the DPU workers can issue the spawning of a follow-up task. These tasks will then first be transferred to the MRAM and then to the CPU, once the DPU execution completes, and spawned there, enabling the control of what is to happen on the CPU in a task driven manner.
Notably the implementation of the task based runtime on the DPU implemented in this thesis, does not support the spawning of new tasks that will be executed on the DPUs during the DPU execution, and instead relies on the CPU for this. The decision to not implement this feature has been made, as it was estimated to not be useful for the aggregation workload this thesis focuses on. Additionally, it would require additional operations for the workers to check for new tasks in some fashion.

Chapter 4

Aggregation on an Upmem PIM system

In this chapter, the implementation of various algorithms for aggregation on the UPMEM PIM system is going to be discussed. The first section of this chapter is going to introduce the given workload that the following sections are going to tackle. The second section is going to evaluate whether, and if how, the algorithms commonly used for multi-core aggregation can be modified to work on the UPMEM PIM system. Finally, the third section is going to discuss the specific implementations that have been developed for these algorithms as well as which parameters are of note for invoking them for different workloads.

4.1 Workload definition

As stated in chapter 1 the goal of this thesis is the investigation of the viability of computational memory for the workload of aggregation when compared to other currently available hardware. While the problem of aggregation on the UPMEM architecture in particular has already been investigated by other authors[9], the important question of the initial location of the table data has been answered with the DRAM of the CPU. While this approach is in line with the usage of the UPMEM PIM system as an accelerator, this thesis is going to specifically investigate the aggregation of data that is already present in the MRAM of the DPUs of an UPMEM system, as would be the case during the runtime of some in memory database application. Additionally this thesis seeks to use a variaty of different algorithmic approaches to investigate UPMEM's viability in a broader context.
In doing so this thesis aims to provide a perspective on the UPMEM PIM architecture that is more accommodating to its architectural shortcomings, in particular its reliance on the limited DRAM bus, in addition to the requirement of having to perform a bit shuffling operation whenever moving data from and to the UPMEM DIMMs[8]. Additionally, this thesis will focus on the aggregation of simple 8 byte tuples made up of a 4 byte key and

a 4 byte value which are stored in a column based arrangement. MRAM tables are set to always hold 32 MB of data i.e. 2^22 tuples.

4.2 Applicability of CPU aggregation algorithms

4.2.1 Sorting and hash based aggregation on UPMEM

As discussed in section 2.1.1 aggregation algorithms generally are either implemented using sorting or hashing. However, sort based aggregation algorithms were usually dismissed due to their longer $O(n \cdot logn)$ runtime as compared to the expected $O(n)$ runtime of hash based algorithms.
While the inherent runtime constraints of sort based algorithms naturally apply on all von Neumann based computer architectures, including that of the UPMEM PIM system, two particular characteristics of it have an additional impact on the viability of these algorithms on it.

1. The lack of direct communication channels between the individual DPUs means, that additional transfers to the CPU are necessary, to merge the result across the DPUs.
2. The comparatively low clock speed of the UPMEM DPUs means that, unlike on a CPU, workloads which require too many instructions per byte may become compute bound, which inherently means, that any workload with more operations per byte is at an additional disadvantage on the UPMEM architecture [8].

Due to the inherent disadvantage the UPMEM architecture therefore has when implementing sorting, sorting based aggregation algorithms were dismissed as possible candidates for the implementation in this thesis. The remainder of this chapter is therefore going to focus on the discussion of hash based aggregation algorithms on the UPMEM PIM system.

4.2.2 Independent and shared table aggregation and hybrid aggregation on UPMEM

This section is going to discuss its three titular aggregation algorithms together, as they all fundamentally represent different ideas on how to handle and/or avoid the contention induced when aggregating values into a hash table, while thereby carrying with them different properties in terms of their cache utilization.

As the MRAM of the UPMEM DIMMs may be compared to the DRAM of a traditional CPU based architecture, while the WRAM shares similarities with a CPUs cache while also being fully controllable through the programmer, the UPMEM PIM architecture offers new ways of implementing these algorithms, while requiring more implementation effort.

This is because, while direct access to the MRAM is programmatically possible, doing so is usually implemented with very small transfer sizes between WRAM and MRAM [9], leading to more time being spent on the access latency[8].
This thesis is therefore going to focus on implementing various versions of these three algorithms and combinations thereof, in order to investigate their performance. The ideas we are going to follow in particular are, where using shared, or independent hash tables is useful or detrimental, i.e. WRAM and/or MRAM, what their appropriate sizes are, to minimize the overhead on the CPU, while still having advantageous properties on the DPUs and whether using the WRAM for the hash tables is beneficial for large group by cardinalities.

4.2.3 Partitioning based aggregation on UPMEM

Section 2.1.5 detailed the properties and use cases of partitioning based aggregation on multi-core CPUs. This section may be summarized as partitioning being the best suited algorithm for very large group by cardinalities while leading to too much overhead for small and medium group by cardinalities.
However while the possible modified algorithms discussed in section 4.2.2 are all expected to not perform very well for large group by cardinalities, the architectural characteristics of the UPMEM DIMMs are particularily disadvantageous for implementing partitioning based aggregation, for two reason which are going to be discussed in the following sections:

Limited memory

Unlike the aggregation step, the partitioning step does not reduce the amount of data between the input and the output. As all algorithms assume that the data that shall be aggregated is present in an unsorted manner in the MRAM of the DPUs, this means that either less than half of the MRAM may be used to store the tables meant to be aggregated, or many invocations of the DPUs are necessary in order to fully partition the data and to transfer it to the CPU, regardless of the group by cardinality.
Therefore any partitioning implementation on UPMEM DPUs would additionally need to be implemented with locks and dynamically sized partitions, increasing the overhead of the partitioning step on the DPUs.

Inter DPU communication

While the factor of the limited memory may lead to additional necessary invocations for the partitioning step, the fact, that these invocations are necessary in the first place is the major issue with partitioning on the UPMEM system. This is because by having to transfer the entire data through the narrow DRAM bus and then having to iterate over all of it in order to convert it from the shuffled version in which it was present on the DPUs fundamentally defeats the purpose of using the UPMEM architecture in the first place.

Advantageous characteristics

While the characteristics of the partitioning phase mentioned above discourage the usage of that algorithm, particularly the access patterns on the DPUs on a micro level are actually more advantageous then those of the directly hash based algorithms. This is because, partitioning on a micro level has the same access patterns on a DPU as partitioning on a multi-core processor:

1. Data is read sequentially from the table, which therefore can be accessed in large transfer sizes from the MRAM. Notably however, this also applies to aggregation without partitioning[7].

2. As partitioned values may also be written sequentially to the end of a WRAM local buffer, which may be evicted into the MRAM upon being full in one piece the transfers to the MRAM can also technically benefit from a large throughput[7].

While the partitioning operation technically offers beneficial characteristics in terms of the access patterns inside of the DPUs, the fact, that practically the entire memory present on the UPMEM DIMMs must be transferred to the CPUs DRAM and processed in addition to having to wait for the completion of the partitioning on the UPMEM DIMMs the partitioning approach was dismissed for the aggregation on the UPMEM DIMMs.

4.3 Implemented algorithms and parameters

For the purpose of this thesis nine variations of the ideas represented by the three algorithms discussed in section 4.2.2 were implemented and evaluated experimentally (Note: the term in parenthesis behind the algorithms is the name with which they will be identified in the graphs later):

1. Independent table aggregation in the WRAM without eviction (WRAM SEPARATE NO EVICT)

2. Shared table aggregation in the WRAM without eviction (WRAM SHARED NO EVICT)

3. Independent table aggregation in the WRAM and insertion of individual keys into independent MRAM tables when full (WRAM SEPARATE SINGLE EVICT MRAM SEPARATE)

4. Independent table aggregation in the WRAM and insertion of individual keys into a shared MRAM table when full (WRAM SEPARATE SINGLE EVICT)
5. Shared table aggregation in the WRAM and insertion of individual keys into a shared MRAM table when full (WRAM SHARED SINGLE EVICT)
6. Independent table aggregation in the WRAM and transfer of entire blocks when full (WRAM SEPARATE TABLE EVICT)
7. Shared table aggregation in the WRAM and transfer of entire blocks when full (WRAM SHARED TABLE EVICT)
8. Independent table aggregation in the MRAM (MRAM SEPARATE)
9. Shared table aggregation in the MRAM (MRAM SHARED)

4.3.1 Independent and shared table aggregation in the WRAM with individual eviction

The three algorithms represented by this section, are for the most part equivalent to the hybrid algorihtm from the CPU. The hybrid algorithm is technically only equivalent to the independent WRAM and shared MRAM algorithm, with its idea of avoiding contention. The other concept followed by the hybrid algorithm is the usability of the cache for common values, which is the main reason, the two other combinations have been implemented for the UPMEM system, as the WRAM is utilized as a pseudo cache by these algorithms. These algorithms therefore work, by first aggregating in the WRAM hash tables, and only if no more space is left, falling back on the larger MRAM hash tables.

4.3.2 Independent and shared table aggregation in the WRAM with block eviction

Unlike the cache the communication between the WRAM and MRAM is not only directly controllable through the application code, but it also allows for variable transfer sizes, of up to $2048B$. The architecture of UPMEM therefore supports using a different approach to the hybrid algorithm. After having already inserted a tuple into the WRAM hash table, where access latencies are short, instead of probing the MRAM with its far slower access latencies[8] individually, when the WRAM is full, these algorithms instead copy whole blocks of data (typically of the size of one of the independent tables in the WRAM) into the MRAM. While doing so promises to significantly improve the runtime on the DPUs, this approach is likely to quickly fill the MRAM with many of these blocks requiring eviction to the CPU DRAM and putting the burden of aggregation on the CPUs instead. The main expected use case of these algorithms is therefore data, where keys are locally clustered, as this would lead to few evictions, while preserving the fast access of the MRAM.

4.3.3 Independent and shared table aggregation in the MRAM

These two algorithms represent are the most comparable to the independent and shared table aggregation algorithms on the CPU, in that they don't use any intermediary tables before that. However, due to the lack of an automated caching system, as present on the CPU, these algorithms are expected to perform poorly for small group by cardinalities, as they cannot utilize the fast access rates of the WRAM. The main purpose of implementing these implementations is to evaluate for large group by cardinalities if avoiding the additional operations associated with the WRAM tables has a performance benefit, or if they actually accelerate the aggregation still.

4.3.4 Independent and shared table aggregation in the WRAM without eviction

Unlike the remaining algorithms, these two are implemented by only using the WRAM for aggregation and copying the result into the MRAM at the end. In doing so, the main purpose of them is to find a benchmark for the peak possible throughput on the UPMEM PIM.
All nine algorithms were implemented using the MxTasking framework, which was extended to be compatible with the UPMEM PIM system as discussed in chapter 3.

4.3.5 Algorithmic decomposition into tasks

As the algorithms where implemented using the MxTasking framework, The algorithms were therefore generally subdivided into the following tasks, which were scheduled onto the DPUs/CPUs respectively:

1. The DPUCleanupTask. This task is responsible for initializing the hash tables, which the algorithm uses. At the end the workers synchronize one another through a barrier.
2. The DPUAggregateTask. This task performs the aggregation as explained in section 4.3.
3. The DPUCopyTask. This task copies the remaining keys (if any) from the WRAM hash table(s) to the MRAM hash table(s).
4. The DPUTransferTask. This task is only spawned for one worker per rank and its only functionality is to spawn the CPUAggregateTask for that rank.
5. The CPUAggregateTask. This task transfers the MRAM hash tables from the DPUs of its rank, to the DRAM of the CPU and aggregates it first into a private hash table. Then it merges that tables content into the result table atomically. It also checks if any of the workers of its rank failed to finish aggregating and initiates a new invocation of the rank if needed.

6. The DPUAggregatePostStopTask. The first aggregation task to be executed on the DPU worker after it failed to aggregate. It is responsible for aggregating the remaining tuples of the failing aggregate task.

7. The DPUCleanupPostStopTask. Only reinitializes the MRAM hash table(s), as its contents have already been read by the CPU. As the aggregation process might have failed before evicting all results from the WRAM, the WRAM tables are not altered.

All of the above DPU tasks with the exception of the DPUAggregateTask/DPUAggregatePostStopTask and the DPUTransferTask are always spawned once per DPU worker. The DPUAggregateTask is typically spawned once per transfer, however in order to evaluate the overhead induced by the large number of necessary tasks (e.g. one rank with $2GB$ of tuples requires $2GB/2KB/task = 1000000 tasks$), super tasks were used, which internally execute multiple DPUAggregateTasks, while only taking up the space of one.

4.3.6 Core loop

All algorithms have been implemented using a similar core aggregation loop. That is, in order to keep the computational load low, linear probing through a continuous hash table was implemented. By avoiding the use of extendable hash tables e.g. through pointers or the like, the transfer of the results back to the CPUs DRAM is also simplified, as no additional communication is necessary to establish the size of the final hash table. Before starting the core loop the DPU worker must transfer a segment of tuple data from the $32MB$ tuple buffer in the MRAM into a private WRAM buffer. This transfer is performed once per task and can be implemented using varying transfer sizes. The performance of the parameters with regard to these transfer sizes will be evaluated in section 5. After that, the worker can start iterating over the tuples present in its buffer and execute the core loop for each of them.
The core component of the loop can be seen in listing 4.1. Every tuple insertion/update is implemented by first computing the hash function once. After that an infinite do while loop is executed in which every iteration probes one slot of the hash table until an appropriate slot has been found or the table has been determined to be too full to continue efficient aggregation.

```
uint32_t iteration = 0;

uint32_t key_to_insert = wram_data_buffer[index]._key;
uint32_t hash = rotating_hash(key_to_insert);

do
{
        uint32_t hash_slot = ((hash + iteration) & hash_func) + hash_offset;
        uint32_t hash_key = wram_hash_table[hash_value]._key;

        if (hash_key == invalid_key)
        {
                wram_hash_table[hash_value]._key = key_to_insert;
                wram_hash_table[hash_value]._val = wram_data_buffer[index]._val;
                break;
        }
        else if (hash_key == key_to_insert)
        {
                wram_hash_table[hash_value]._val += wram_data_buffer[index]._val;
                break;
        }
        else
        {
                ++iteration;
        }
}
while (true);
```

Listing 4.1: The central code of the aggregation algorithms on the UPMEM DPUs, that is executed for every key when aggregating in seperate hash tables.

Each probing step therefore consists of 2 case distinctions, which separate into three cases, as can be seen in listing 4.1:

1. No key is present, meaning that the current key can be inserted.
2. The same key is present, meaning that the slot must be updated.
3. A different key is present, meaning that probing must continue.

Depending on which of the nine algorithms described above is used, locking may be employed, and if the algorithm allows the eviction into the MRAM in case of a full hash table one of the eviction policies which will be described in section 4.3.7 will be used.

4.3.7 Eviction policy

Unlike the remaining seven algorithms the first two algorithms have been implemented to only work when the group by cardinality is sufficiently small so that all keys fit into the applicable hash tables in the WRAM, as data is only moved to the MRAM upon the completion of all aggregation operations, so that the CPU has access to them. The main purpose of this is to benchmark the overhead caused by the check of the fill rate.
Therefore the remaining seven algorithms all must perform some kind of check for whether the currently used hash table is full or not. Because both the WRAM and the MRAM hash tables are limited in size, this applies to both types of hash tables. For example, in the case of algorithm 3 from the list above, if a DPU worker determines that it cannot

insert the current key into its private WRAM hash table because it is full, it would then abort the probing process in the WRAM and compute the hash for the key in the MRAM. After that, it would start probing its private MRAM hash table for an appropriate slot. If it were to determine that its MRAM hash table is full as well, it would write data to the $Early_Stop_Info$ struct, and stop operating entirely. The CPUAggregateTask responsible fro retrieving and accumulating the data on the CPU would then check if any DPU worker had to abort the aggregation by retrieving and evaluating the $Early_Stop_Info$ data from its rank and initiate a new invocation of the DPUs by creating and spawning new tasks for the DPUs.

In order to determine whether a hash table is full two approaches were implemented. They will be discussed in the following sections.

Check on probing step

One way of implementing a heuristic for determining the appropriate point in time for evicting a key, is to check after every probing step whether a set number of probing iterations has been performed while trying to aggregate the current tuple and to abort the aggregation if that is the case. The key chosen for eviction then is the one that was first visited during probing for the current tuple to insert. This is done, because having probed for a set amount of slots, means that, that many slots were occupied in a row. Using the assumption, that the slot at the beginning of the chain of full slots is possibly the oldest one, this method can function as a heuristic for finding older keys in the table to evict, without requiring the maintenance of an additional data structure.

The new key is then inserted in the slot of the old one. While this might seem contradictory to the assumption made for the heuristic, as the new key will then be at the beginning of a chain with the given length, the likelihood of the same key being evicted is relatively low, as evictions may only take place for keys that were in the position a new key has been mapped onto by the hash function initially.

Check on insert

Even though, the probing based eviction method is relatively simple, the fact, that this method requires the DPU worker to check the number of probing steps taken for every probing step may cause a big overhead, if the number of average probing steps taken during aggregation is high. Therefore, an alternative eviction policy was implemented. This policy is based on the actual number of different keys present in the hash table.

It has been implemented, by incrementing a counter for the number of keys in the table, whenever a key is inserted into the hash table. If the counter reaches a limit set relative to the size of the hash table (we use 75% of its capacity) an eviction will be triggered. However, unlike the probing based policy, as evictions now may occur on the first probing step, while the currently probed slot is always empty, there is now no candidate available in an immediate sense. Therefore, in order to avoid having to maintain an additional data structure for finding a candidate, an additional linear probing process is performed starting from the slot where the tuple was inserted, in order to find any occupied slot which is then picked as the slot to evict.

4.3.8 Other implemented parameters

In addition, to the different eviction policies the following parameters were implemented, of which some will be experimentally evaluated in chapter 5:

- Different hash functions implemented through bit operations based on [10].
- Varying tuple transfer sizes.
- Varying WRAM and MRAM hash table sizes.
- Varying MRAM bucket sizes.

Chapter 5

Experiments

This chapter is going to discuss the experimental setup and the procured results of the various aggregation algorithms that have been discussed in the chapters 2 and 4. It should be noted, that all results discussed in the upcoming sections are the arithmetic mean across five separate runs.

5.1 Experimental setup

The experiments regarding the UPMEM PIM systems were carried out on a two socket server equipped with two Intel(R) Xeon(R) Silver 4216 CPUs which are each equipped with 8 cores clocked at 2.1GHz. This system is also equipped with a 256GB RAM in addition to the 40 ranks of DPUs distributed across 20 DIMMs. Notably out of the 40 present UPMEM ranks in this configuration only 30 have the full advertised 64 DPUs available, with only 2546 out of 2560 DPUs working. This is because 10 of the present ranks have faulty DPUs which appears to be a common occurrence stemming from the new manufacturing process [8].
As section 5.3.1 is going to be concerned with a discussion of the selection of parameters for the efficient execution of the aggregation algorithm on the DPUs, the experiments there are going to be carried out using only 1 rank. This is because these experiments are exclusively concerned with the runtime on the DPUs which is independent of concerns of transfer times and the following merging of the results on the CPU. Sections 5.3.2 and 5.3.3 on the other hand are concerned with evaluating the performance of the entire workflow of the aggregation on the DPUs and the following merging operation on the CPU. Therefore these experiments were carried out using all 30 fully functional ranks that were available in the used installation.

As mentioned in section 4.1, it should be noted that all table data is assumed to be already present in the MRAM of the UPMEM DPUs at the start of the aggregations. Therefore,

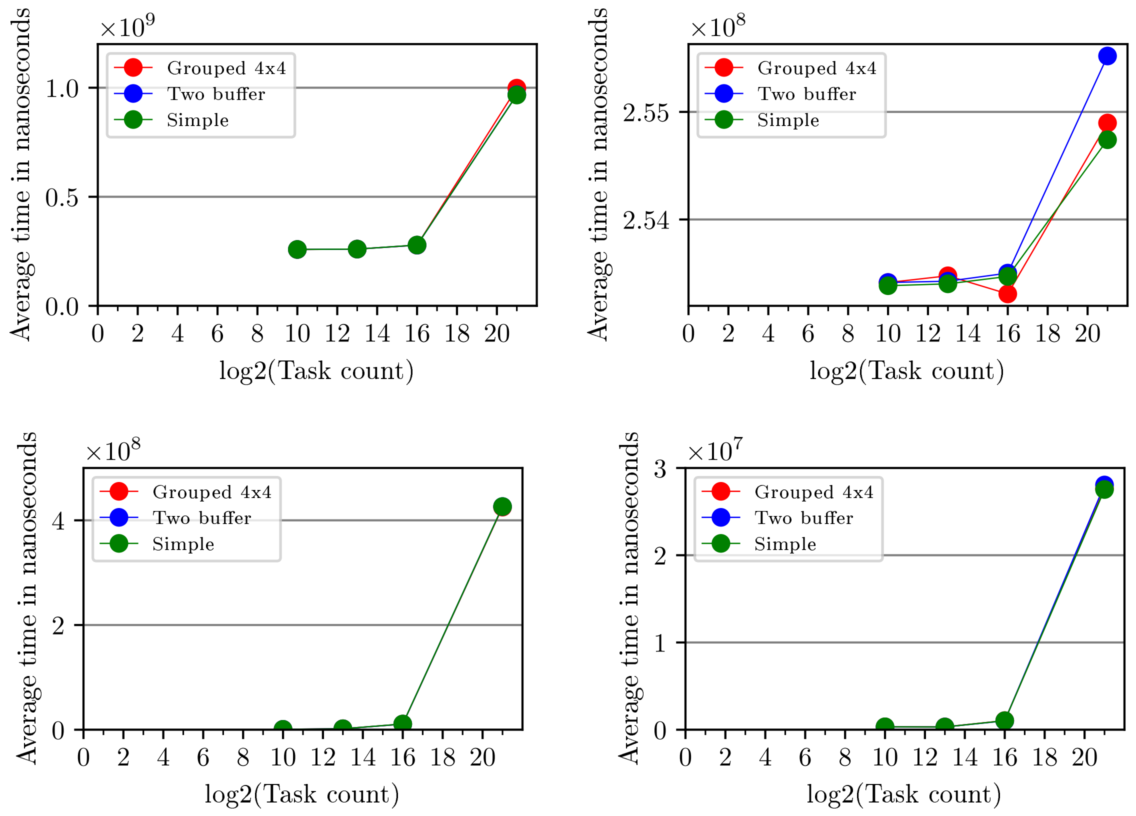


Figure 5.1: The different runtimes of an aggregation algorithm using independent WRAM hash tables, for the three different runtime implementations over a rank, for different task counts.
Top left: Total runtime. Top right: DPU runtime.
Bottom left: CPU task generation time. Bottom right: Task transfer to DPU time.
Note: The task count represents only the number of physically present tasks. The actual workload is the same (2GB of data), as tasks have merely been merged into super tasks, due to the repetitive nature of the workload.

all measurements are concerned with the time starting from the moment the first task is being created up to the moment the results of the aggregation have been combined in one CPU hash table.

The experiments regarding modern multi-core CPUs were carried out on a two socket server equipped with two Intel(R) Xeon(R) Gold 6230 CPUs which are each equipped with 20 cores clocked at 2.1 GHz. This system is also equipped with a 192 GB RAM.

5.2 Performance of th MxTasking runtime extension

This section is going to discuss the results of the experiments regarding the performance of the different MxTasking runtime implementations on the DPUs. As the runtime of each DPU is functionally independent of those on other DPUs these experiments were conducted using only one rank and one CPU core. The different runtime implementations were tested for a selection of task counts ranging from 1024 to 2097152, while implementing the same workload, in order to evaluate the overhead caused by each task in the context of its used DPU runtime implementation. The aggregation operator evaluated for these workloads, was the sum operator on 8 byte tuples consisting of a 4 byte key and a 4 byte value.

As can be seen in figure 5.1 (top left), the use of more tasks for the DPU control incurs a significant overhead increasing the total execution time by a factor of 4 across all DPU runtime implementations. When inspecting the individual components of the runtime however, it becomes clear, that the actual DPU runtime is barely affected by the increase in the number of tasks, differing only by roughly 1% (see figure 5.1 (top right)).

Figure 5.1 (bottom left) shows, that the majority of this additional time is spent on the task generation on the CPU. This indicates, that the chosen implementation of the task processing on the CPU might be inadequate, as it relies on the use of the relatively large and complex task class already implemented on the CPU, while also using polymorphism to determine the DPU task time, thereby incurring an additional overhead. It might therefore make sense to change the process for processing DPU tasks on the CPU, to immediately using the simpler representation which is also used on the DPU, in the future. It should be noted however, that such a more specialized approach, would lead to tasks no longer being able to be scheduled to either device.

For now, however this thesis is going to proceed by using only super tasks, merging all aggregation tasks per DPU worker into one, for evaluating the experiments in the following sections, as the very uniform property of the aggregation workload allows for this.

Regarding the performance of the different DPU runtime implementations on the DPU, while the performance differences between the three discussed implementations are so small, with respect to the total runtime, it can be seen, that the simple implementation generally outperforms the other two. This implies, that the overhead stemming from the synchronization performed by them is larger, than the time saved by avoiding more frequent access latencies through larger transfers.

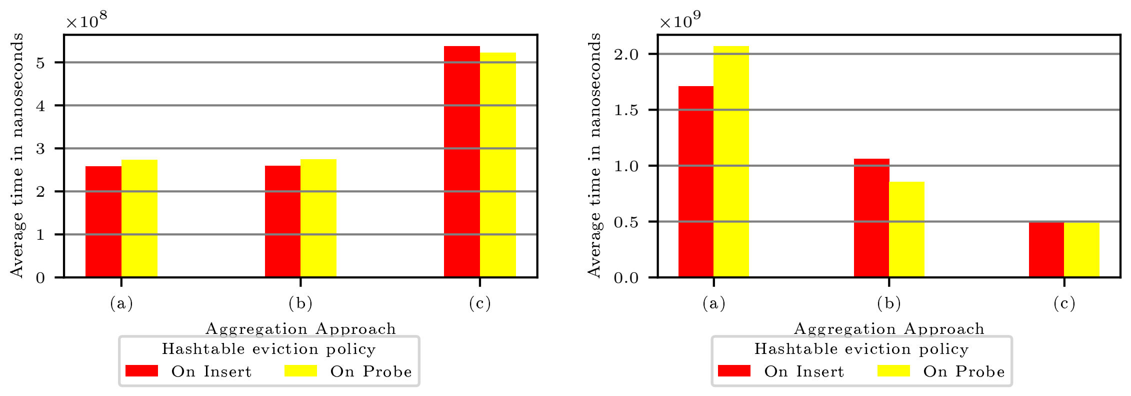


Figure 5.2: The DPU execution time in nanoseconds for the different eviction policies and a group by cardinality of 64(left) and 256(right).
(a): Independent WRAM tables with single eviction into independent MRAM tables
(b): Independent WRAM tables with single eviction into a shared MRAM table
(c): Shared WRAM tables with single eviction into a shared MRAM table

5.3 Aggregation on the UPMEM PIM system

5.3.1 Experimental evaluation of the parameters

Eviction heuristics

The performance of the two eviction detection methods described in section 4.3.7 was experimentally evaluated for the three algorithms which implement the eviction of individual keys. The experiments were performed for two different group by cardinalities, one where they require 50% of the WRAM hash tables space and one where they would require 200% of the WRAM hash tables space therefore forcing eviction.

As can be seen in figure 5.2 the method that checks on every probing step is more performant for the two algorithms that use independent tables in the WRAM, while the method that checks when inserting is more performant for the algorithm that uses a shared table in the WRAM. There is however one exception, which is that the algorithm that uses independent tables in the WRAM but a shared table in the MRAM, switches around for large group by cardinalities (i.e. such that force it to evict to the shared MRAM table) to being faster when checking for every probing step.
One possible explanation for this behavior is, the additional overhead caused, by the method that checks on insertion. While the probing based check method has to execute its check operation on every probing step, the computational work required for it is very low as it is a singular integer comparison. The following eviction is also fast, because it simply chooses the current key.

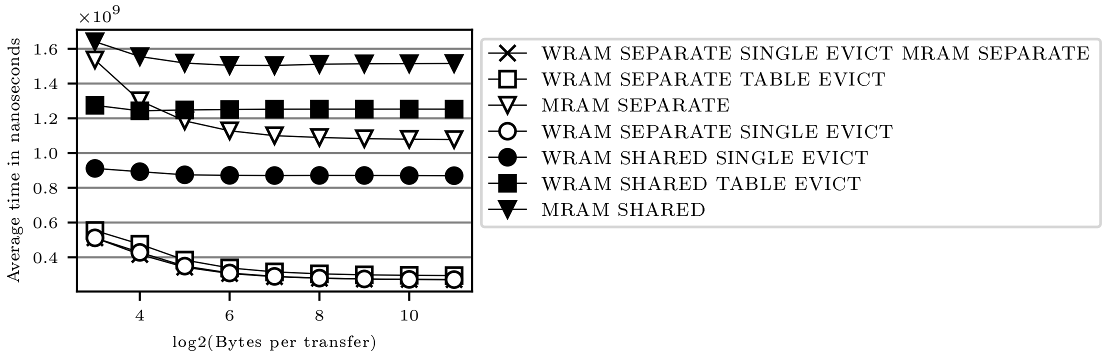


Figure 5.3: DPU execution times for transfer sizes ranging from 8B to 2048B.

The insertion based check method also uses a singular integer comparison to check the fill state of the table. As the hash table stores no information with regard to fitting eviction candidates, the process of finding one is quite expensive, as it requires probing the table until a key is found. It may therefore be, that while the independent hash tables can amortize these costs, as insertions happen rarely, the additional cost for shared tables induced by the additional lock acquisitions for that purpose outweighs the benefit of not having to perform the comparison on every probing step.

Therefore the insertion based check method is going to be used for independent table algorithms, while the shared table algorithms are going to be used in combination with the probing based check method in the comparative experiments of sections 5.3.2 and 5.3.3.

Hash table and transfer sizes

As code that runs on the DPUs may only directly work on data present in the WRAM and as small transfers from and to the MRAM incur a large access latency [8] the 64 KB of available WRAM memory are a contested resource, because dedicating a segment of memory to the function of a buffer for MRAM data transfers allows for a direct increase in performance. For hash based aggregation workloads in particular there are two uses for the memory which compete for the limited WRAM memory, they utilize the memory in such a way that it cannot be shared with the other use. The additional factor of having to use at least 11 tasklets, also means, that this memory requirement has to be multiplied by the number of tasklets.

WRAM tuple buffer

The first use is the WRAM resident tuple buffer. While the data in the WRAM tuple buffer remains static for each transfer from the MRAM, only if it is sufficiently large, can the full bandwidth of the MRAM be utilized[8]. Therefore, increasing the size of the buffer has a direct influence on the possible throughput of any aggregation algorithm.

The result of the experimental evaluation for the different transfer sizes can be seen in figure 5.3. The various algorithms have been evaluated for transfer sizes ranging from 1 tuple i.e. 8 bytes per transfer to 256 tuples i.e. the maximum transfer size of 2048 bytes. While the execution time varies by up to a factor of 2 for the different transfer sizes, thereby justifying the use of larger transfer sizes, the effectiveness of the increase decreases for very large transfer sizes. Therefore, in order to utilize an effective transfer size while still preserving memory for other uses in the WRAM, using a transfer size of 64 tuples has been chosen. In doing so $16 \cdot 64 \cdot 8bytes = 8192bytes$ of the WRAM are used for the transfer buffers only.

WRAM hash table

The second use is the WRAM resident hash table. Unlike the tuple buffer, the hash tables size is restricted to a power of two as not choosing that as its size would disable the use of a bit shift to enforce the mapping of keys into the hash tables boundaries. As divisions are very slow on the UPMEM DPUs[8] this is not an option this thesis considers. Notably this memory segment is also not needed by the pure MRAM aggregation algorithms, though these algorithms are expected to be only advantageous for very large group by cardinalities in which case, most keys will likely not reside in the WRAM for long anyway.
On the other hand the algorithms which do utilize WRAM hash tables are expected to be highly sensitive to the size of those tables. This is because while group by cardinalities are smaller than the WRAM hash table size, workers will access the MRAM exclusively for the retrieval of new tuples into the WRAM tuple buffer, in which case transfer sizes may be relatively large, depending on the choice of that parameter. For larger group by cardinalities however, the algorithms using single tuple eviction will be forced to evict tuples into the MRAM hash table(s). As small transfer sizes lead to the overhead of the access latency becoming a large factor in the total DPU runtime, dedicating a larger amount of memory to the WRAM hash table(s) makes it so that larger group by cardinalities can avoid these expensive accesses.

As can be seen in figure 5.4 the performance impact of the WRAM hash table size is significant and also significantly larger, than that of the transfer size, as the results there only differed by a factor of at most 2 whereas here it differs by factors between 1.5 and 14. The reason for this large impact is likely, that unlike the transfer of the tuple from the MRAM into the WRAM buffer, which only occurs once per tuple, the worker may need to

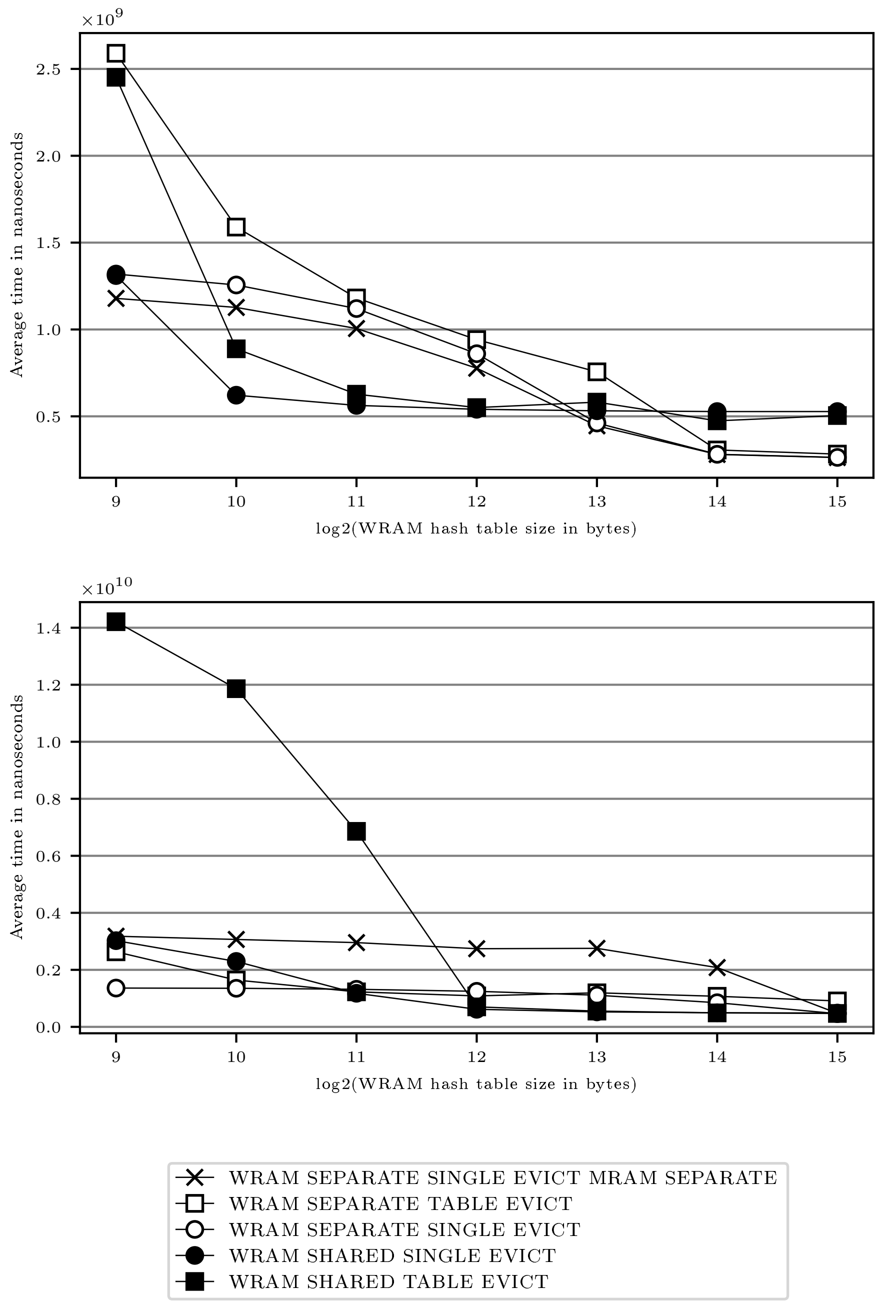


Figure 5.4: DPU execution times for total WRAM hash table sizes ranging from 512B to 2^{15}B, given the group by cardinality 64(top) and 256(bottom).

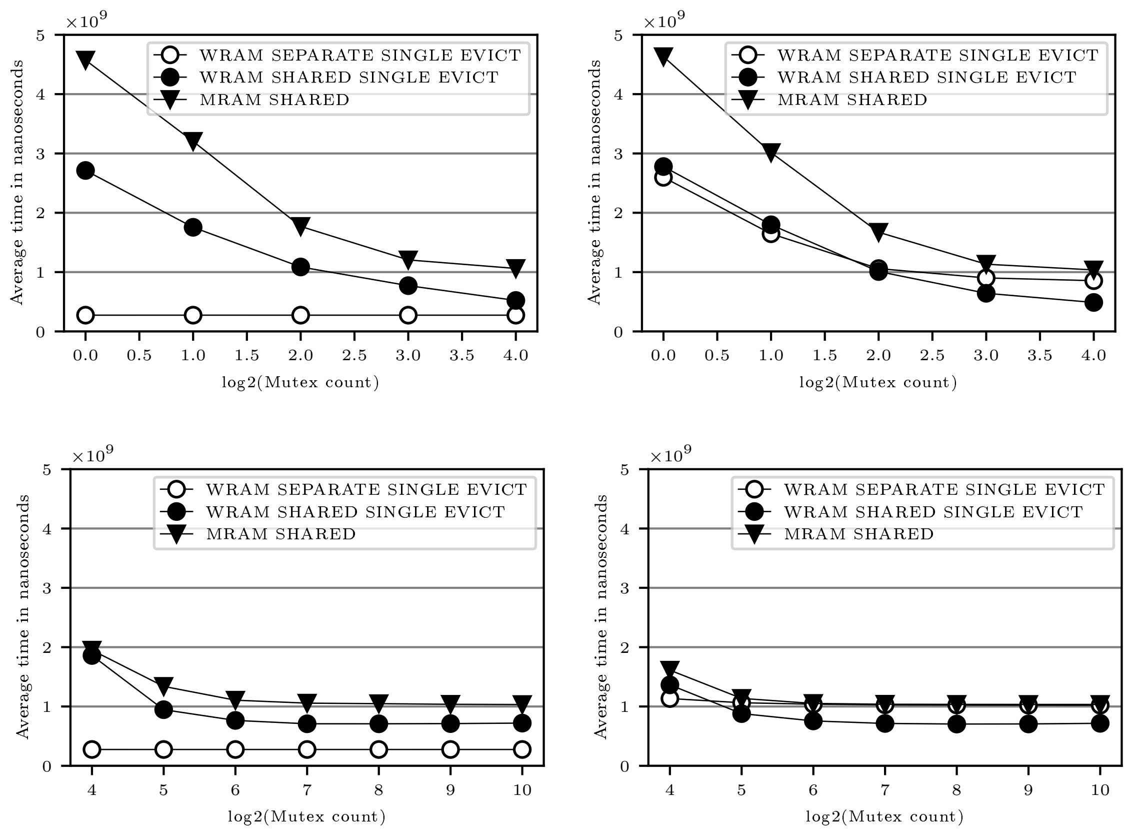


Figure 5.5: The DPU execution time in nanoseconds for the locking implementations and a group by cardinalitiy of 64(left) and 256(right).
Top: DPU execution time for the hardware mutexes.
Bottom: DPU execution time for virtual mutexes using 16 hardware mutexes internally.

probe the hash table many times until it finds a viable slot. As the WRAM hash table size is therefore significantly more impactful for the performance of the algorithms, the decision has been made, to use as much of the WRAM as possible for storing these hash tables. This means, as their size is limited to being a power of two, that the hash tables are set to use at most 32 KB such that space still remains for the stack and transfer buffers. The 32KB result in 256 slots for each independent and 4096 for the shared WRAM hash table.

Locking

This section is going to evaluate the results of the experiments related to finding the best performing locking method for the algorithms utilizing a shared hash table for probing.
As mentioned in section 2.3.1 the UPMEM DPUs support the use of mutexes which are implemented by continuously attempting to change a bit in a special atomic register until the bit has been flipped and the lock acquired. Because each DPU only contains one such

register which has 56 bits, the number of so called hardware mutexes is also limited to 56 per DPU [16]. However, in order to support the shared access to larger data structures the UPMEM SDK also implements so called virtual mutexes, which are implemented by using a user definable number of hardware mutexes to protect the access to the actual mutexes which are implemented as a bit array stored in the WRAM. The Upmem SDK documentation claims therefore that by using virtual mutexes with a sufficient amount of underlying bits, that it is possible to reduce the amount of contention during the critical segment of the code[16].

However, it should be noted, that a hardware mutex pool lock takes 1 assembler instructions to execute, while a virtual mutex takes at least 10[16]. Therefore, the question this thesis aims to answer with the following experimental evaluation is, whether the performance increase created by decreasing the contention during the critical segment of the code outweighs the constant overhead of using the more complex virtual mutexes.

Figure 5.5(top) shows the performance of the hardware mutexes for a mutex count ranging from 1, i.e. effective serialization of the critical section, up to 16, the maximum amount possible when both WRAM and MRAM based tables require locking, due to the maximum number of usable hardware mutexes being 56. Figure 5.5(bottom) shows the performance of the virtual mutexes for a hardware mutex count of 16 and virtual mutex counts ranging from 16 to 1024.

It should be noted that the independent WRAM hash table algorithm with single key eviction is unaffected by the mutexes for the two plots using a low group by cardinality distribution, because the keys can remain WRAM resident there, where locking is not performed in this algorithm.

As can be seen in the figures, the hardware mutexes outperform the virtual ones for the shared WRAM hash table algorithm as well as the independent WRAM hash table algorithm with single key eviction into a shared MRAM table. For the aggregation in shared MRAM tables however, both algorithms show almost the same performance. This is likely because the relative length of the critical section of the MRAM algorithm, caused by the longer MRAM accesses leads to the contention on the 16 mutexes being so high that avoiding it with the slower but more granular virtual mutexes has the same performance impact. The reason, why the algorithm using shared MRAM tables, but independent WRAM tables does not show this effect, in spite of also accessing the MRAM hash table in a shared manner, is likely, that the majority of operations still occur in the WRAM for the given group by cardinalities, thereby reducing the access frequency in the MRAM and thereby the contention.

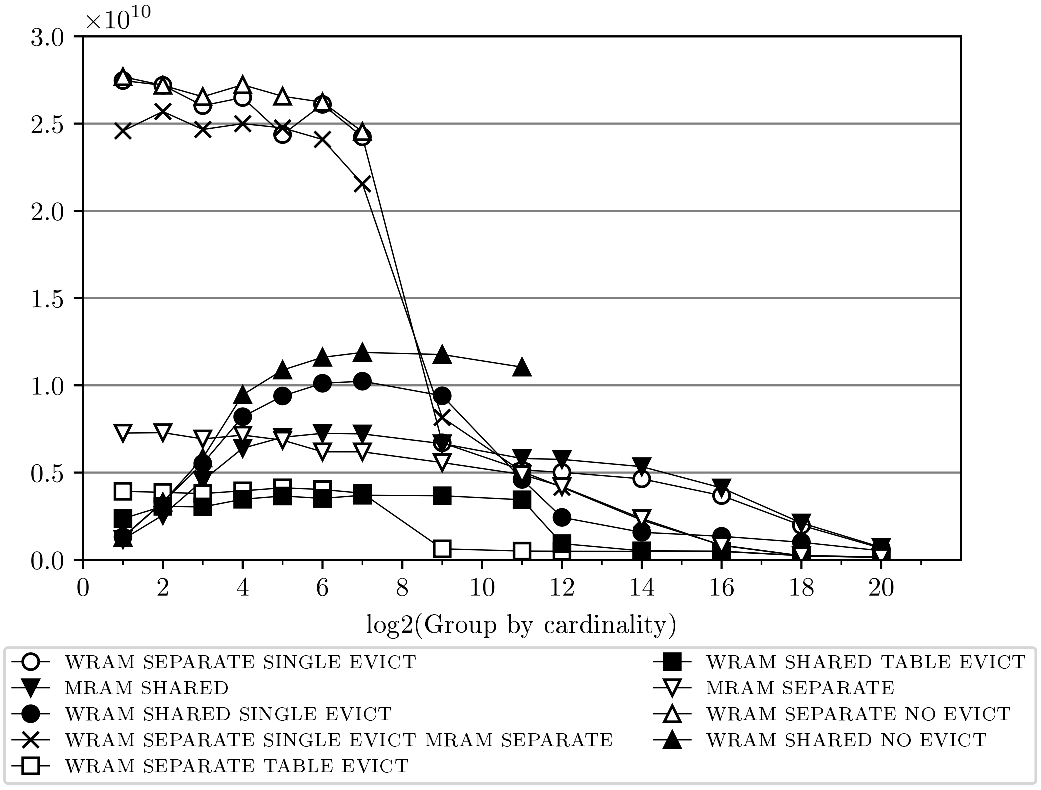


Figure 5.6: Total tuple throughput per second for the various DPU algorithms and group by cardinalities from 2 to 2^{20}

Because the performance of the hardware mutex implementation was better for almost all algorithms and equivalent for the rest, it is selected as the used synchronization tool for the experiments in sections 5.3.2 and 5.3.3.
It should be noted however, that it is likely, that these observations would not apply when the aggregation operation is more complex, as the discussed operations here are only concerned with one value row.

5.3.2 Performance for different group by cardinalities

This section is going to discuss the performance of the different aggregation algorithms for group by cardinalities ranging from 2^1 to 2^{20}, for which the evaluated throughput in tuples per second can be seen in figure 5.6. Additionally, this section is limited to an in-depth discussion of the results for a uniform data distribution.
It should be noted, that for these experiments, the total MRAM size has been set to twice the size of the group by cardinality, however for the algorithms that use independent tables in the MRAM it has been set to 32 times the group by cardinality, because all 16 workers need this much space each. For large group by cardinalities, the maximum total size of the

MRAM hash table slots has been set to 2^{21}, because of the MRAM's space constraints. For the algorithms, that use block eviction, their tendency to evict very often, once the group by cardinality is larger than the hash table size, was used as a reason to statically set the MRAM hash table buffer size for them to 2^{19}, in order to avoid to many evictions to the CPU.

The observable results show in many ways trends which correspond to the trends that were present in the experiments of [6] and [18]. While the results for the independent WRAM table algorithms in the low group by cardinality ranges between 2 and 128, i.e. the cardinalities, where all keys still fit into each of the independent hash tables for the used WRAM hash table size of 256 slots, once it's capacity is reached and the used eviction policy must evict the keys into the MRAM for fill rates of above 75%, the throughput drops from approximately $2.5 \cdot 10^{10}$ down to $0.75 \cdot 10^{10}$ indicating the common small MRAM transfers as a big performance bottleneck.

This assumption is supported by the fact, that these algorithms fall down to a throughput similar to that of the pure MRAM based independent hash table algorithm.

The performance of the shared WRAM hash table algorithms also largely corresponds to those from [6], however unlike those results, the relative performance decrease if compared to the peak performance for a group by cardinality of 512 is not as severe as for the referenced experiments, as throughput only decreases by a factor of 10. This is because, while the locking operations effectively serialize the aggregation, unlike a CPU implementation, where the serialization applies across all CPUs, the serialization on the DPU implementation can only apply within a DPU, thereby still allowing for the massive parallelism enabled by the use of 64 DPUs per rank with 30 ranks.

So for medium group by cardinalities, i.e. 512 onward, the shared algorithms mostly outperform the independent variants. Initially, there is a small range between group by cardinalities of 512 and 2048, where the throughput difference between the independent and shared WRAM table algorithms is relatively large. This is the range where the property of the WRAM tables being shared, means that aggregation can take place entirely within the faster WRAM for larger group by cardinalities, as $256 \cdot 16 = 4096$. For larger group by cardinalities however, even the shared WRAM algorithms cannot keep all keys in the WRAM.

Notably for group by cardinalities larger than 4096 another trend can be observed. Here the algorithms, that use independent hash tables in the MRAM show a significant throughput decrease, which is then followed by the shared MRAM algorithms for even larger group by cardinalities. The same trend occurs for block evict algorithms for even smaller group by cardinalities.

This trend becomes understandable when investigating the decomposition of the time spent per algorithm on the individual operations necessary for performing the aggregation operation. That is, specifically the times spent on the DPU execution, the data transfer times from the DPUs to the CPU DRAM and the time spent on the CPU's aggregation per core, as can be seen in figure 5.7.

While figure 5.7(top) explains the relatively weak performance of the shared WRAM table algorithms for small group by cardinalities, the DPU execution times alone do not explain the poor performance of the independent MRAM algorithms and block evict algorithms for large groupy by cardinalities, as they belong to the best performing algorithms across all group by cardinalities in that respect. In order to understand their poor performance, one must investigate the transfer times and in particular the CPU execution times in figure 5.7(center and bottom). As can be seen there, the independent MRAM algorithms start spending tens of seconds on the aggregation on the CPUs. This is because the maximum MRAM space available for the hash tables is 16MB, as the tuples take up 32MB in addition to the tasking infrastructure, making $2^2 4$ the largest usable power of 2 of bytes for the hash tables. This means, that each independent hash table has only space for 2^{17} slots, so starting from those group by cardinalities the independent MRAM hash table algorithms will have to often evict the table to the CPUs DRAM. Therefore, this trend follows the same pattern as the one observed for the WRAM hash tables, only with a far larger impact on the performance, as each interruption forces a complete stop of the aggregation on a rank, and the core responsible for the CPUAggregateTask, aggregate the entire MRAM hash table content an additional time.

5.3.3 Performance for different group by cardinalities given different data distributions

While the previous experiments give a good initial insight into the performance of the UPMEM PIM system for the aggregation workload, this section is going to discuss a selection of different data distributions for which the different aggregation algorithms were used for a range of group by cardinalities. For this sake a selection of the distributions, which were also discussed in [6] were chosen, more specifically an input sorted by the keys, an input with a heavy hitter distribution, in this case one where one of the keys represent 50% of the input, a sequential distribution. Additionally, a moving cluster distribution was used for the experiments, using a window width of 32 for the sliding window.

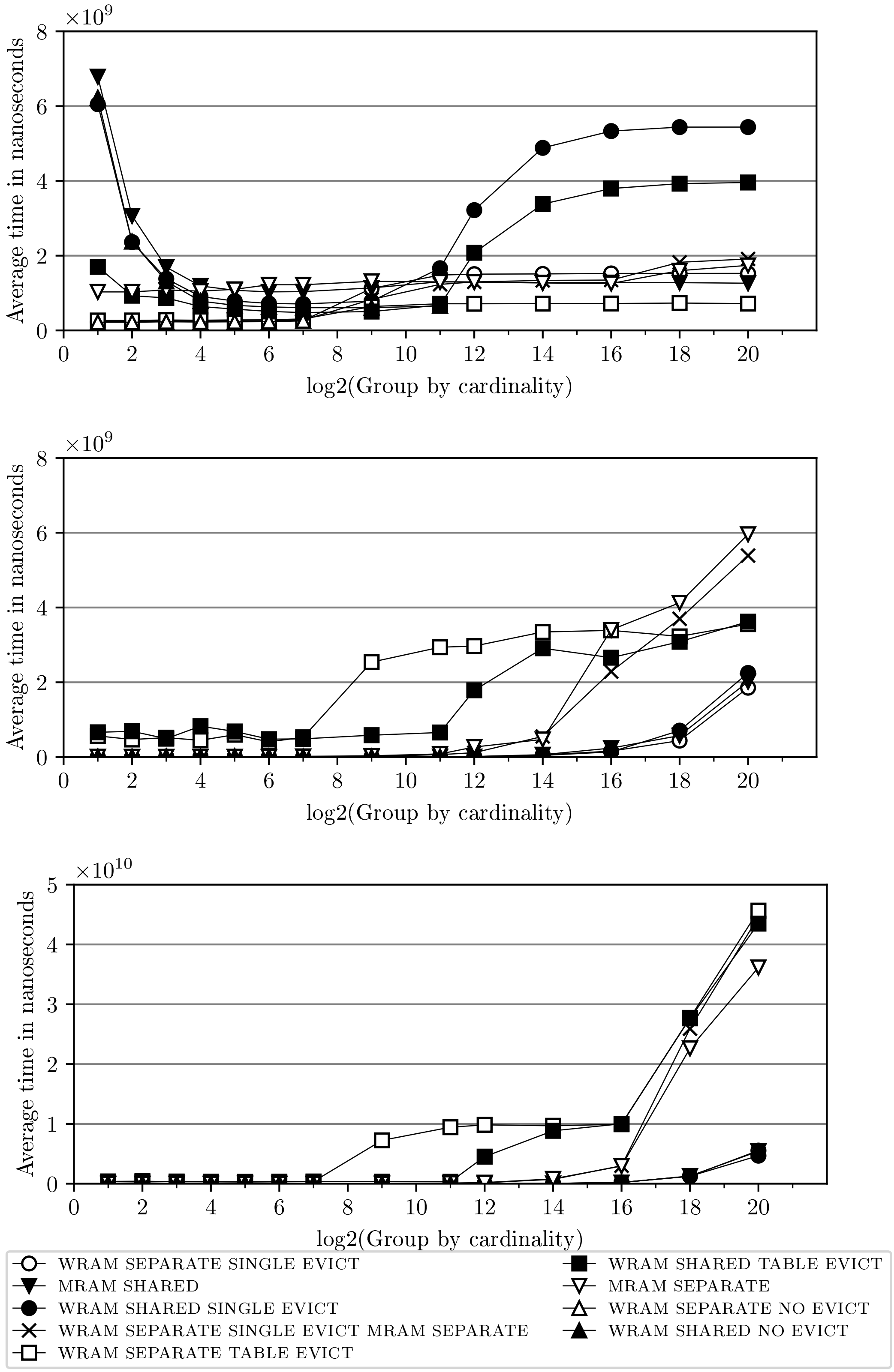
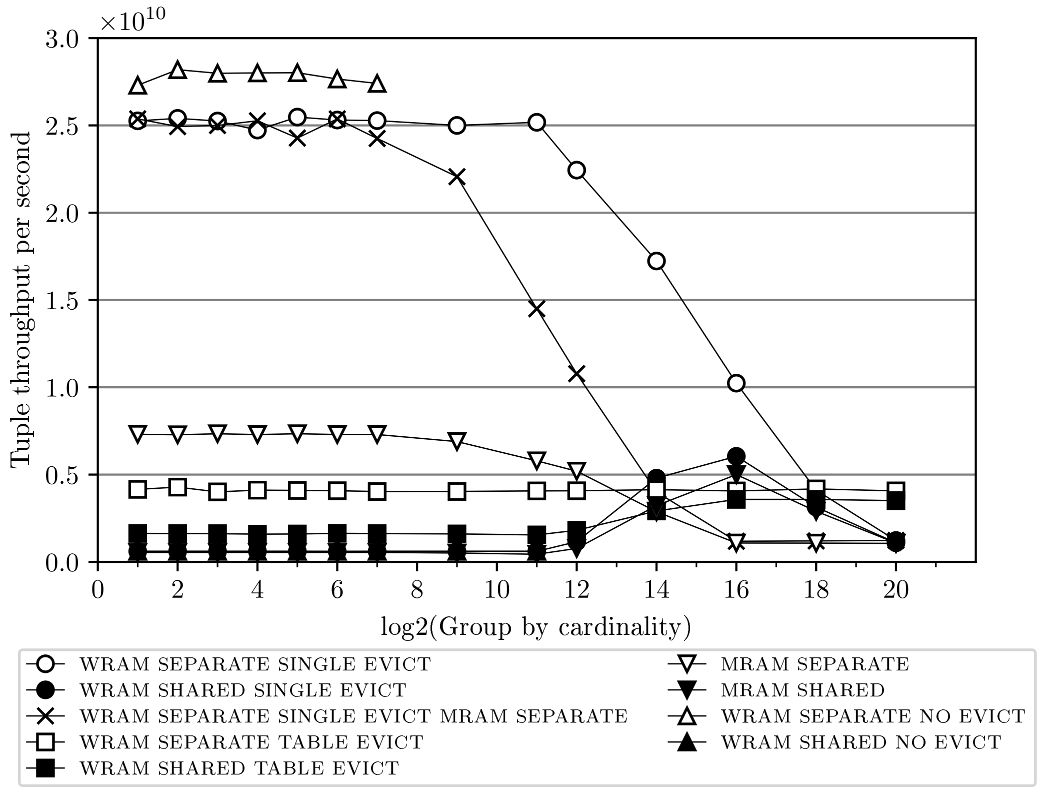


Figure 5.7: DPU times(top), data transfer times from DPU(center) and CPU aggregation times(bottom) for the various DPU algorithms in nanoseconds and group by cardinalities from 2 to 2^{20}



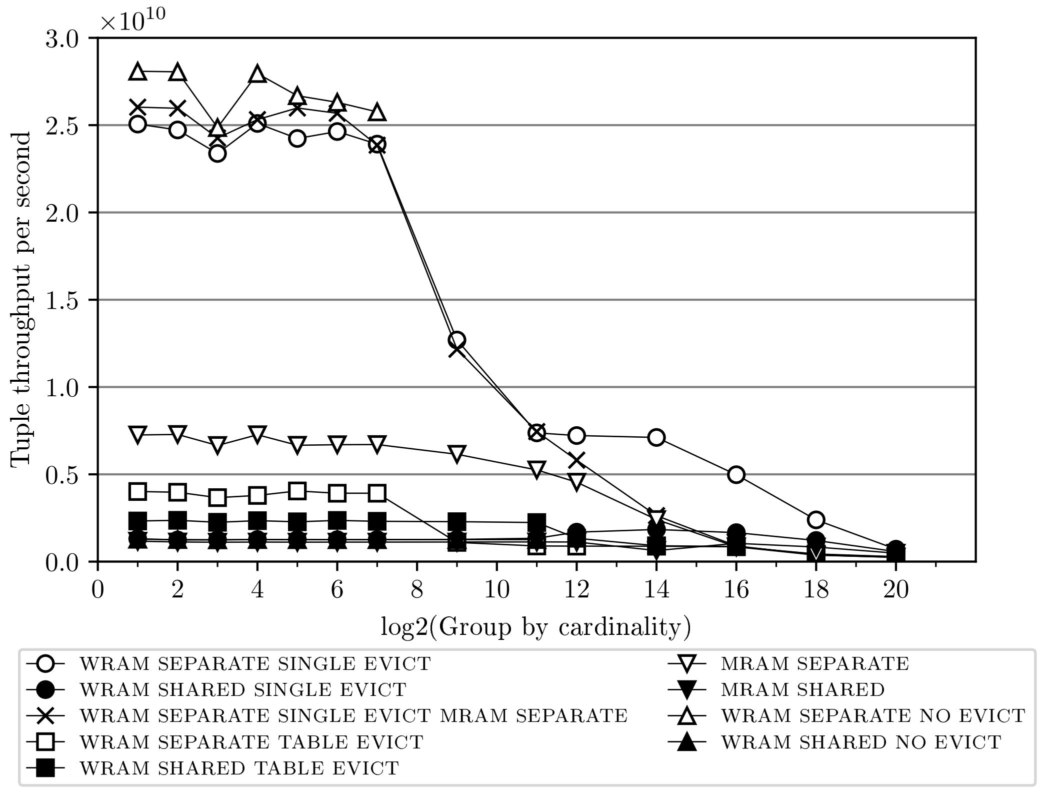


Figure 5.8: Tuple throughput per second for a sorted input(top) and a heavy hitter distribution(bottom) group by cardinalities from 2 to 2^{20}

Sorted input

Figure 5.8(top) shows the results of the experimental evaluation for the sorted input data. The main difference in the observed performance, is that all shared table algorithms perform significantly worse for this distribution, than they did for the uniform data. While the contention induced overhead, that was observed for low group by cardinalities and uniform data only applied for group by cardinalities of up to 16, for the sorted input, a performance increase can only be seen for these algorithms starting from group by cardinalities of 2^{12}. Notably no comparable effect was observable for the observations of [6] when using sorted inputs.

This difference can however be explained when one analyzes the layout of the sorted tuples across the DPUs. As the experiments have been conducted using $30 \cdot 64 = 1920$ DPUs sorting the input for group by cardinalities smaller than that size means, that every DPU will only contain one key which corresponds with the value from which the shared table algorithms start showing a better performance. The reason why [6] did not observe this behavior for their CPU implementation is, that while the data was distributed among 32 cores, the effects of contention observed for other distributions also mostly occurred for group by cardinalities of that scale, thereby hiding it. It should also be noted, that while this effect decreases the performance of the shared table algorithms for small and medium group by cardinalities, for even larger group by cardinalities, i.e. those equal to the number of workers, it can guarantee the absence of contention.

An observation that is in line with those made in [6] is that the performance of independent table algorithms stays at a very high level for far larger group by cardinalities. The reason for this is once again the local effect across the DPUs. Because of this, even the smaller capacity of the independent hash tables in the WRAM of 256 should suffice for global group by cardinalities of up to $256 \cdot 1920 < 2^{19}$. However, while the performance stays high for slightly larger group by cardinalities than in the case of the uniform distribution, i.e. 128 vs. 512 (independent WRAM independent MRAM) and 2048 (independent WRAM shared MRAM), the effect does seemingly not apply for the predicted group by cardinalities.

The reason for this is however not the DPUs themselves, but rather the chosen MRAM hash table sizes. As discussed at the beginning of section 5.2, they have been chosen, to always be double the size of the group by cardinality for each hash table, with 2^{21} being the maximum regardless of the previous rule. Therefore, the observed slow-down can be explained by the increasing transfer times to the CPU, even if most of the hash table is unused.

What also stands out for the sorted input, is that the algorithm with independent WRAM tables transferring large blocks when evicting keys, performs very consistently across the

different group by cardinalities, which is likely due to the fact, that the workload represented by a sorted input is very well suited for this algorithm. This is because a sorted input guarantees that a key x will never occur again after another key y has been read after it. This means that keys evicted due to lack of space, must only ever be evicted once. As the block evict algorithms are particularly well fit to exploit this kind of workload, as they can benefit from their larger transfer sizes during eviction, without filling the MRAM quickly and forcing evictions to the CPU.
The reason, why this algorithm does not outperform the individual eviction based algorithms therefore, is likely the large chosen MRAM size discussed in section 5.2, causing longer transfer times and thereby decreasing the throughput.

Heavy hitter distribution

The performance of the algorithms for the heavy hitter distribtion can be seen in figure 5.8(bottom). The performance of the algorithms for this distribution is similar to that for the uniform distribution, with two notable exceptions.

1. Unlike the uniform distribution, the performance of the shared table algorithms stays poor even for large group by cardinalities, which is explained by the consistent contention on the heavy hitter key for 50% of all tuples. This behavior was also noticeable in the results of [6].

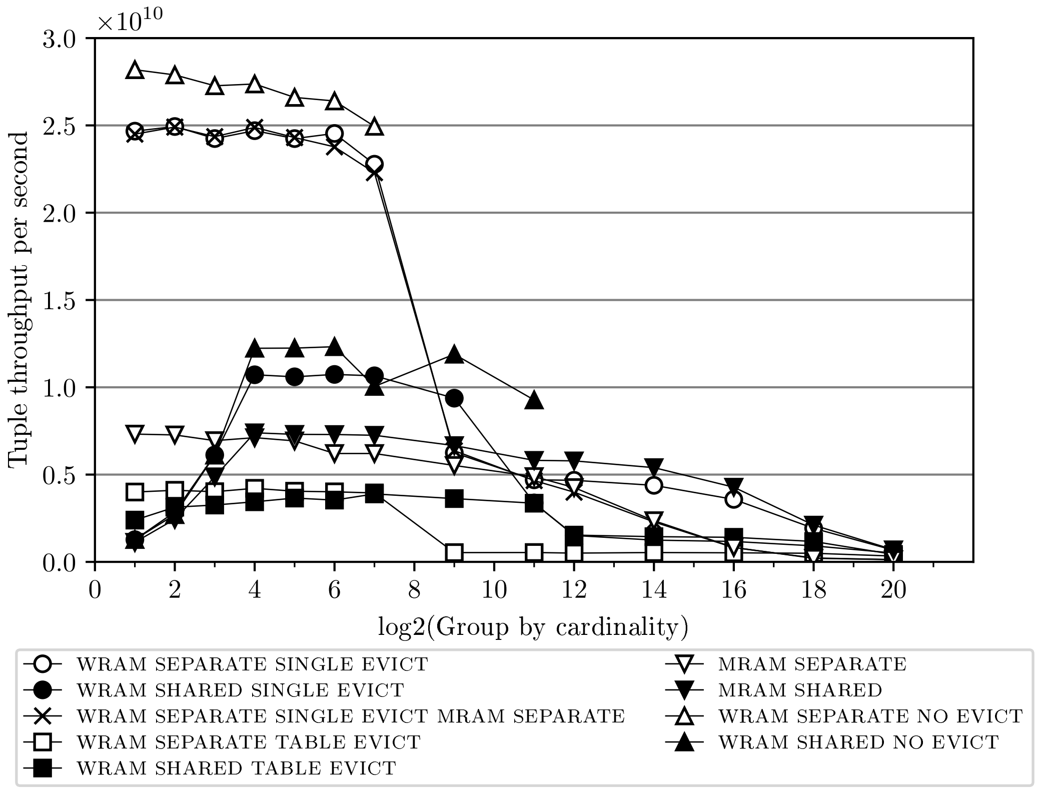
2. Both the individual eviction algorithms using independent WRAM tables perform better for medium group by cardinalities of 512 to 2048. This is likely due to the fact, that they can benefit from keeping the heavy hitter in the WRAM, and thereby having significantly improved access times for 50% of the input, while avoiding the contention on it.

Sequential input

The algorithms show performance characteristics for the sequential input, that are almost identical to those for the uniform distribution. This is because, while the pattern presented by the sequential input is very predictable, neither the algorithms nor the UPMEM architecture provide a way to exploit it, which means, that the only relevant attribute of the distribution for the performance is its local group by cardinality. And as that is functionally identical to that of a uniform distribution, that is what determines the results seen in figure 5.9(top).

Moving cluster distribution

Unlike the other distributions discussed in this chapter this distribution was only used for a group by cardinality range from 64 to 2^{20}, because the distribution with a window width



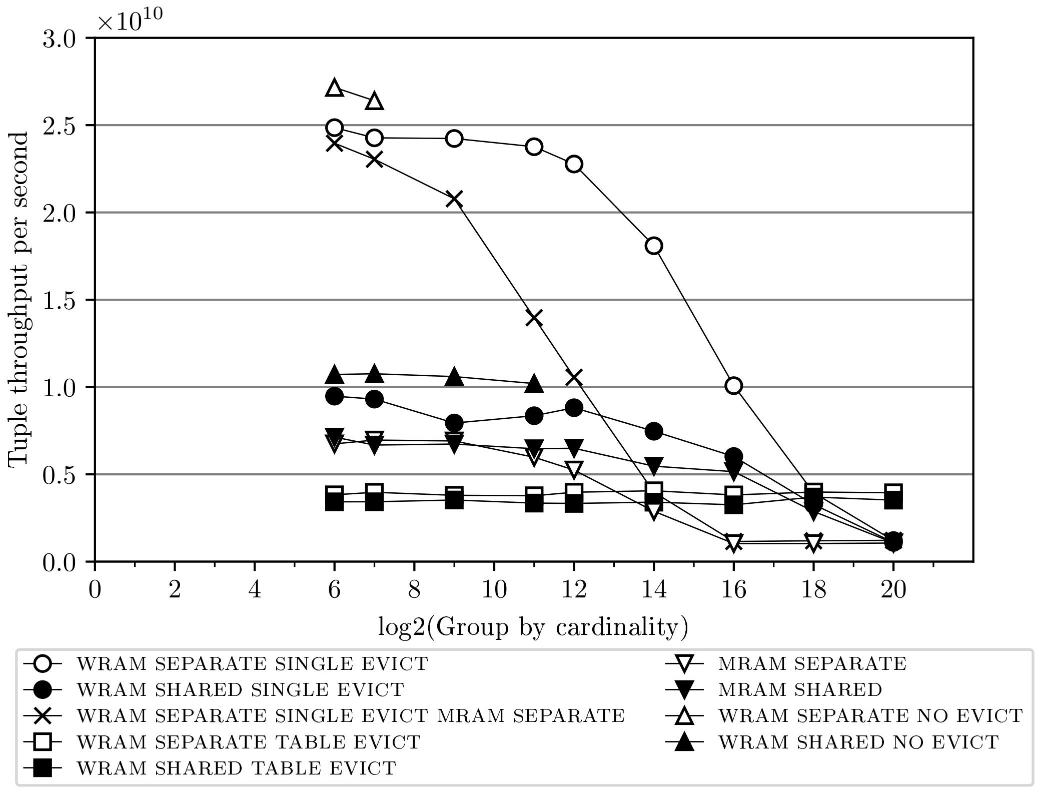


Figure 5.9: Tuple throughput per second for a sequential input(top) and a moving cluster distribution(bottom) group by cardinalities from 2 to 2^{20}

of 32 would break down to a uniform distribution for smaller group by cardinalities [6]. As can be seen in figure 5.9(bottom), the performance of almost all algorithms is significantly better than for other algorithms. In spite of the locally similar keys, the shared table algorithms perform well for this distribution, as the locally present group by cardinality of 32 is sufficient for the avoidance of most contention, while allowing for the need for evictions to be rare, as the nature of the distribution as a sliding window across the different keys, means that this distribution has similar characteristics to the sorted distribution. While keys are not guaranteed to not reappear once they have been evicted, it is far less likely to happen, than for uniform distributions.

These local low group by cardinality for stretches of the input, has the same effect as the sorted input had on the independent hash table algorithms and block evict algorithms, leading to the evictions of keys into the MRAM to be rare. It should be noted however, that this is likely due to the choice of the used window width of 32 fitting well into any WRAM hash table.

5.4 Aggregation on a modern multi-core CPU

So far the discussion of the experimental results in this thesis has solely focussed on the performance of the aggregation operation on the UPMEM PIM system for different parameters and workloads. This section is going to compare these results with results which have been produced on a modern multi-core CPU in order to procure insights into the viability of the UPMEM PIM system in a productive context.

5.4.1 Performance for different group by cardinalities

For this purpose figure 5.10 shows the results of the DPU based algorithms (in black) overlaid with the results of implementations of the CPU based algorithms discussed in section 2.1(in red).

While the general trends of the results from the current CPU correspond with those from [6], the performance of the independent table approach is higher than that of the shared tables for far larger group by cardinalities. While one might assume that this is due to the larger cache sizes on the modern hardware, the size of the LLC for the used system only has a capacity of $27.5MB$, while the memory requirement for storing every hash table independently for 20 cores is around $640MB$ for a group by cardinality of 2^{21}, thereby exceeding the cache size by a factor of 23. It must therefore be assumed that the overhead of the atomic operations on this platform is so great that it exceeds that of the direct RAM access, as the effects of contention should no longer be that great for this group by cardinality.

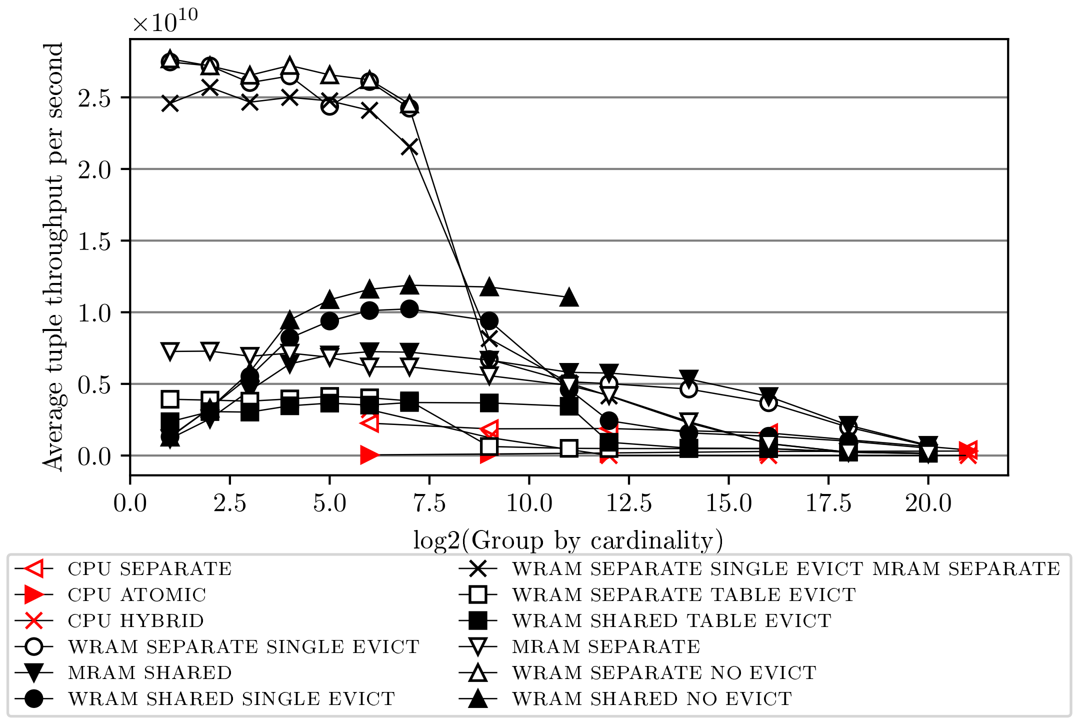


Figure 5.10: Average tuple throughput per second for the DPU(black) and CPU(red) algorithms for group by cardinalities from 2 to 2^{20}

When comparing the peak throughput of the modern CPU implementation ($3 \cdot 10^9$ tuples per second) with that of the UPMEM PIM system ($2.75 \cdot 10^{1}0$) for a uniform distribution, it can be seen, that the UPMEM PIM system outperforms the CPU implementation by a factor of almost 10. This can be explained by the significantly larger number of operations per second the 30 used ranks can perform ($30 \cdot 64 \cdot 350$MHz = 672GOps) when compared to the 20 CPU cores ($20 \cdot 4.1$GHz= 82 GOps), in addition to the fact, that the individual throughput of a DPUs when accessing the MRAM in an optimal way is $628MB/s$, which is multiplied for every DPU, as they do not share a bus[8].
For large group by cardinalities however, the performance of the UPMEM PIM system is almost identical to that of the CPU implementation. This is because while the separation of the workload into the locally responsible DPUs allows for the massive parallelism that enabled the previously discussed peak throughput, the larger result size for large group by cardinalities, means, that each DPU must transfer those results to the CPU's DRAM. This simultaneously puts a strain on the limited bus, while also forcing the CPU cores to perform the merge operation across all hash tables off its rank. As can be seen in figure 5.7(bottom) this leads to the majority of the time being spent not on the DPUs, but on the CPU cores.

Chapter 6

Conclusion

6.1 Summary

Throughout this thesis, the algorithmic problem of aggregation was examined in a context of two types of modern computational hardware. Additionally, the task based MxTasking framework was extended to enabling the scheduling of tasks onto an UPMEM PIM system on a tasklet granularity.

The performance of the UPMEM system was consistently able to outmatch that of a modern multi-core processor for group by cardinalities of up to 2^{20}. The maximum achievable performance is however not only highly dependent on the structure of the data, as locally clustered distributions of data allow for a higher throughput even for larger group by cardinalities, it also requires knowledge on the appropriate parameter and algorithmic choices to exploit this structure.

For large group by cardinalities, i.e. those starting from 2^{20}, the inherent structure of the UPMEM PIM system, i.e. the isolated nature of the DPUs, with their inability to communicate with one another, leads to the necessity of a large amount of communication between the CPU cores and the DPUs, shifting the majority of the time spent from the DPUs to the CPU cores and significantly decreasing performance.

With regard to the utilization of the UPMEM PIM system through a task based framework the following can be said. While different approaches were presented, aimed at combating the overhead induced by the management of the tasks on the DPUs, it turned out, that the impact of these tasks was negligible on the DPUs themselves ($\approx 1\%$).

The overhead on the CPU side of the MxTasking runtime was however severe, indicating the need for a different approach to this implementation, or the shift of more of these structures to the DPUs.

6.2 Future Work

While this thesis attempted to evaluate the impact of modern hardware on the algorithmic problem of aggregation, it did so only through the two hardware platform types of multi-core processors and computational memory. While multi-core processors represent a large component of the modern hardware landscape, their raw computational power is limited compared to that of modern hardware that is dedicated to massively parallel computation such as GPUs. Work therefore remains to be done in evaluating how these platforms compare to the results obtained in this thesis.

Though this thesis did examine the performance impact of the implementation of a task based framework on a computational memory platform, the uniform nature of the code executed in the implemented aggregation workloads has not provided many opportunities for the use of some of the more advanced features usable on task based software platforms, such as features for communication between the DPU workers within the runtime. Therefore, the behavior of the platform for more heterogeneous workloads remains to be seen. In this regard however, the overhead of a very fine granular task based control flow must be taken into consideration, as this thesis already showed the performance impact it may have when implemented naively.

Appendix A

List of additional used tools and resources

- The extension of the MxTasking framework has been implemented using the MxTasking framework version available at https://git.cs.tu-dortmund.de/DBIS/MxTasking, using commit d36d7850246791f3f72e29bde99ffec4acda470e as a code basis, and thereby also some of its dependencies.
- The benchmarks used the arg_parse library available at https://github.com/p-ranav/argparse for the parsing of the command line arguments.
- Unsourced graphics have been generated using either tikz/tikz-uml, matplotlib or OpenOffice Draw.
- This thesis has been authored using latex and in particular the following latex packages:
  - listings
  - xcolor
  - enumerate
  - algorithmic
  - hidelinks
  - layouts
  - pgf
  - graphicx
  - subfigure
  - pdfpages
  - tikz

- ifthen
- xstring
- calc
- pgfopts
- tikz-uml

List of Figures

Listings

Bibliography

[1] Ailamaki, Anastassia, David J. DeWitt, Mark D. Hill and David A. Wood: *DBMSs on a Modern Processor: Where Does Time Go?* In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[2] Baumstark, Alexander, Muhammad Attahir Jibril and Kai-Uwe Sattler: *Adaptive Query Compilation with Processing-in-Memory.* In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 191–197, 2023.

[3] Baumstark, Alexander, Muhammad Attahir Jibril and Kai-Uwe Sattler: *Processing-in-Memory for Databases: Query Processing and Data Transfer.* In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, page 107–111, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Bernhardt, Arthur, Andreas Koch and Ilia Petrov: *pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories.* In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, page 44–52, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Chang, Kevin K.: *Understanding and Improving the Latency of DRAM-Based Memory Systems.* CoRR, abs/1712.08304, 2017.

[6] Cieslewicz, John and Kenneth A. Ross: *Adaptive aggregation on chip multiprocessors.* In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 339–350. VLDB Endowment, 2007.

[7] Cieslewicz, John and Kenneth A. Ross: *Data partitioning on chip multiprocessors.* In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, page 25–34, New York, NY, USA, 2008. Association for Computing Machinery.

[8] Gómez-Luna, Juan, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira and Onur Mutlu: *Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture.* CoRR, abs/2105.03814, 2021.

[9] König, Maximilian: *Query Processing on Computational Memory*, December 2023.

[10] Mailund, Thomas: *The Joys of Hashing: Hash Table Programming with C.* 01 2019.

[11] Mühlig, Jan and Jens Teubner: *MxTasks: How to Make Efficient Synchronization and Prefetching Easy.* In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1331–1344, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Mühlig, Jan and Jens Teubner: *Micro Partitioning: Friendly to the Hardware and the Developer.* In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, DaMoN '23, page 27–34, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Mutlu, Onur, Saugata Ghose, Juan Gómez-Luna and Rachata Ausavarungnirun: *A Modern Primer on Processing in Memory.* CoRR, abs/2012.03112, 2020.

[14] Polychroniou, Orestis and Kenneth A. Ross: *High throughput heavy hitter aggregation for modern SIMD processors.* In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Redlich, Tobias: *Task-Verarbeitung in heterogenen Umgebungen*, May 2022.

[16] SAS, UPMEM: *User Manual - UPMEM DPU SDK 2023.2.0 Documentation*, 2023. `https://sdk.upmem.com/2023.2.0/` [Accessed: 15.04.2024].

[17] Wirth, Niklaus: *Tasks versus Threads: An Alternative Multiprocessing Paradigm.* Softw. Concepts Tools, 17(1):6–12, 1996.

[18] Ye, Yang, Kenneth A. Ross and Norases Vesdapunt: *Scalable aggregation on multicore processors.* In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, page 1–9, New York, NY, USA, 2011. Association for Computing Machinery.